EDICIÓN 20 ANIVERSARIO

El programador pragmático

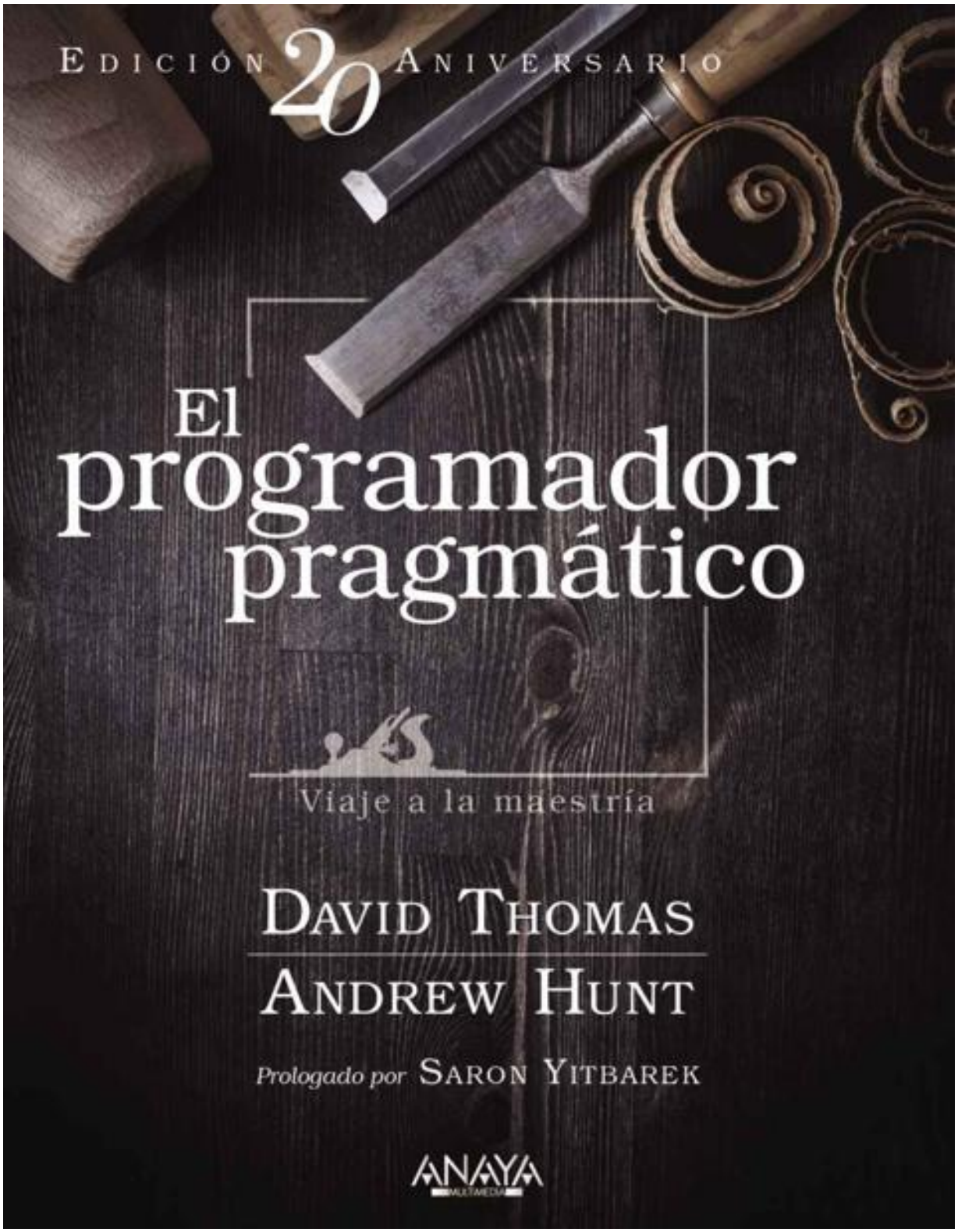
Viaje a la maestría

DAVID THOMAS

ANDREW HUNT

Prologado por SARON YITBAREK

ANAYA MULTIMEDIA


Edición 20 Aniversario
El programador pragmático
Viaje a la maestría
David Thomas
Andrew Hunt
Prologado por Saron Yitbarek
ANAYA MULTIMEDIA

Para Juliet y Ellie,
Zachary y Elizabeth,
Henry y Stuart

Agradecimientos de la segunda edición

Hemos disfrutado de literalmente miles de conversaciones sobre programación en los últimos 20 años, hemos conocido a muchas personas en conferencias, cursos y, a veces, incluso en el avión. Cada una de ellas ha aportado algo a nuestra comprensión del proceso de desarrollo y ha contribuido a las actualizaciones de esta edición. Gracias a todas esas personas (y seguid avisándonos cuando nos equivoquemos).

Gracias a los participantes en el proceso beta del libro. Vuestras preguntas y comentarios nos han ayudado a explicar las cosas mejor.

Antes de pasar a la fase beta, compartimos el libro con algunos colegas para recibir sus opiniones. Gracias a VM (Vicky) Brasseur, Jeff Langr y Kim Shrier por sus comentarios detallados y a José Valim y Nick Cuthbert por las revisiones técnicas.

Gracias a Ron Jeffries por dejarnos utilizar el ejemplo del *sudoku*.

Eterna gratitud al personal de Pearson que accedió a dejar que creásemos este libro a nuestra manera.

Un agradecimiento especial a la indispensable Janet Furlow, que domina todo lo que toca y nos mantiene a raya.

Y, por último, una mención a todos los programadores pragmáticos del mundo que han estado haciendo que la programación sea mejor para todo el mundo durante los últimos veinte años. Por otros veinte más.

Sobre los autores

David Thomas y **Andrew Hunt** son dos de las voces más conocidas y prestigiosas en la comunidad de desarrollo de software. Ponentes y conferenciantes por todo el mundo. Juntos, fundaron la colección Pragmatic Bookshelf, que publica libros de vanguardia galardonados para desarrolladores de software. También fueron dos de los firmantes y autores del *Manifiesto Ágil*, considerado la biblia de las metodologías ágiles y del desarrollo de software.

Dave actualmente enseña en la universidad, tornea madera y juega con nuevas tecnologías y paradigmas.

Andy escribe ciencia ficción, es un músico activo y le encanta jugar con la tecnología.

Índice

Agradecimientos de la segunda edición

Sobre los autores

Prólogo

Prefacio de la segunda edición

- Cómo se organiza el libro
- ¿Qué hay en un nombre?
- Código fuente y otros recursos

Del prefacio de la primera edición

- ¿Quién debería leer este libro?
- ¿Qué hace a un programador pragmático?
- Pragmatistas individuales, equipos grandes
- Es un proceso continuo

1. Una filosofía pragmática

- 1 Es su vida
  - Las secciones relacionadas incluyen
- 2 El gato se comió mi código fuente
  - Confianza del equipo
  - Asuma la responsabilidad
  - Las secciones relacionadas incluyen
  - Retos
- 3 Entropía del software
  - Primero, no haga daño
  - Las secciones relacionadas incluyen
  - Retos
- 4 Sopa de piedras y ranas hervidas
  - El lado de los aldeanos

Las secciones relacionadas incluyen
Retos
5 Software lo bastante bueno
Implique a sus usuarios en la compensación
Saber cuándo parar
Las secciones relacionadas incluyen
Retos
6 Su cartera de conocimientos
Su cartera de conocimientos
La creación de su cartera
Objetivos
Oportunidades para el aprendizaje
Pensamiento crítico
Las secciones relacionadas incluyen
Retos
7 ¡Comuníquese!
Conozca a su público
Sepa lo que quiere decir
Elija el momento
Elija un estilo
Dele un buen aspecto
Involucre a su público
Sepa escuchar
Responda a la gente
Documentación
Resumen
Las secciones relacionadas incluyen
Retos

2. Un enfoque pragmático

8 La esencia del buen diseño
ETC es un valor, no una regla
Las secciones relacionadas incluyen
Retos
9 DRY: los males de la duplicación

DRY es más que código
Duplicación en el código
Duplicación en la documentación
Duplicación relativa a la representación
Las secciones relacionadas incluyen
10 Ortogonalidad
¿Qué es la ortogonalidad?
Beneficios de la ortogonalidad
Diseño
Herramientas y bibliotecas
Creación de código
Pruebas
Documentación
Vivir con ortogonalidad
Las secciones relacionadas incluyen
Retos
Ejercicios
11 Reversibilidad
Reversibilidad
Arquitectura flexible
Las secciones relacionadas incluyen
Retos
12 Balas trazadoras
Código que brilla en la oscuridad
Las balas trazadoras no siempre dan en el blanco
Código trazador versus creación de prototipos
Las secciones relacionadas incluyen
13 Prototipos y notas en post-its
Cosas para las que puede crear prototipos
Cómo utilizar prototipos
Crear prototipos de arquitectura
Cómo no utilizar prototipos
Las secciones relacionadas incluyen
Ejercicios
14 Lenguajes de dominio
Algunos lenguajes de dominio del mundo real

Características de los lenguajes de dominio
Compensaciones entre lenguajes internos y externos
Un lenguaje de dominio interno de bajo coste
Las secciones relacionadas incluyen
Retos
Ejercicios
15 Estimaciones
¿Cuánta exactitud es suficiente exactitud?
¿De dónde vienen las estimaciones?
Estimar el calendario de los proyectos
Qué decir cuando le pidan una estimación
Las secciones relacionadas incluyen
Retos
Ejercicios

3. Las herramientas básicas

16 El poder del texto simple
¿Qué es el texto simple?
El poder del texto
Mínimo común denominador
Las secciones relacionadas incluyen
Retos
17 Jugar con el intérprete de comandos
Su propio intérprete de comandos
Las secciones relacionadas incluyen
Retos
18 Edición potente
¿Qué significa “fluidez”?
Avanzar hacia la fluidez
Las secciones relacionadas incluyen
Retos
19 Control de versiones
Empieza en la fuente
Ramificaciones
Control de versiones como centro de proyectos

Las secciones relacionadas incluyen
Retos
20 Depuración
Psicología de la depuración
Mentalidad de la depuración
Por dónde empezar
Estrategias de depuración
Programador en tierras extrañas
La búsqueda binaria
El elemento sorpresa
Lista de comprobación de depuración
Las secciones relacionadas incluyen
Retos
21 Manipulación de texto
Las secciones relacionadas incluyen
Ejercicios
22 Cuadernos de bitácora de ingeniería
Las secciones relacionadas incluyen

4. Paranoia pragmática

23 Diseño por contrato
DBC
Implementar DBC
DBC y fallo total temprano
Invariantes semánticas
Contratos dinámicos y agentes
Las secciones relacionadas incluyen
Retos
Ejercicios
24 Los programas muertos no mienten
La captura y liberación es para la pesca
Fallo total, no basura
Las secciones relacionadas incluyen
25 Programación asertiva
Aserciones y efectos secundarios

Deje las aserciones activadas
Las secciones relacionadas incluyen
26 Cómo equilibrar los recursos
Anidar asignaciones
Objetos y excepciones
Equilibrio y excepciones
Cuando no podemos equilibrar los recursos
Comprobar el equilibrio
Las secciones relacionadas incluyen
Retos
Ejercicios
27 No vaya más rápido que sus faros
Cisnes negros
Las secciones relacionadas incluyen

5. O se adapta o se rompe

28 Desacoplamiento
Train Wrecks
Los males de la globalización
La herencia añade acoplamiento
De nuevo, todo tiene que ver con el cambio
Las secciones relacionadas incluyen
29 Malabares con el mundo real
Eventos
Máquinas de estados finitos
El patrón Observer
Publish/Subscribe
Programación reactiva, streams y eventos
Los eventos son ubicuos
Las secciones relacionadas incluyen
Ejercicios
30 Transformar la programación
Encontrar transformaciones
¿Por qué es esto tan genial?
¿Qué pasa con el manejo de errores?

Las transformaciones transforman la programación
Las secciones relacionadas incluyen
Ejercicio
31 Impuesto sobre la herencia
Algo de historia
Problema de usar la herencia al compartir código
Las alternativas son mejores
La herencia rara vez es la respuesta
Las secciones relacionadas incluyen
Retos
32 Configuración
Configuración estática
Configuración como servicio
No escriba código dodo
Las secciones relacionadas incluyen

6. Concurrencia

Todo es concurrente
33 Romper el acoplamiento temporal
Buscar la concurrencia
Oportunidades para la concurrencia
Oportunidades para el paralelismo
Identificar oportunidades es la parte fácil
Las secciones relacionadas incluyen
Retos
34 Estado compartido es estado incorrecto
Actualizaciones no atómicas
Transacciones de recursos múltiples
Actualizaciones no transaccionales
Otros tipos de accesos exclusivos
Doctor, me hago daño...
Las secciones relacionadas incluyen
35 Actores y procesos
Los actores solo pueden ser concurrentes
Un actor simple

Sin concurrencia explícita
Erlang prepara el escenario
Las secciones relacionadas incluyen
Retos
36 Pizarras
Una pizarra en acción
Los sistemas de mensajería pueden ser como pizarras
Pero no es tan simple...
Las secciones relacionadas incluyen
Ejercicios
Retos

7. Mientras escribe código

37 Escuche a su cerebro reptiliano
Miedo a la página en blanco
Luche contra sí mismo
Cómo hablar lagarto
¡Hora de jugar!
No solo su código
No solo código
Las secciones relacionadas incluyen
Retos
38 Programar por casualidad
Cómo programar por casualidad
Cómo programar de forma deliberada
Las secciones relacionadas incluyen
Ejercicios
39 Velocidad de los algoritmos
¿Qué queremos decir con estimar algoritmos?
Notación Big O
Estimación con sentido común
Velocidad de los algoritmos en la práctica
Las secciones relacionadas incluyen
Retos
Ejercicios

40 Refactorización
¿Cuándo deberíamos refactorizar?
¿Cómo refactorizamos?
Las secciones relacionadas incluyen
41 Probar para escribir código
Pensar en las pruebas
Escritura de código guiada por pruebas
TDD: necesita saber adónde va
De vuelta al código
Pruebas unitarias
Pruebas en relación con el contrato
Pruebas ad hoc
Construya una ventana de pruebas
Una cultura de pruebas
Las secciones relacionadas incluyen
42 Pruebas basadas en propiedades
Contratos, invariantes y propiedades
Generación de datos de prueba
Encontrar asunciones malas
Las pruebas basadas en propiedades nos sorprenden a menudo
Las pruebas basadas en propiedades también ayudan al diseño
Las secciones relacionadas incluyen
Ejercicios
Retos
43 Tenga cuidado ahí fuera
El otro 90 %
Principios de seguridad básicos
Sentido común vs. criptografía
Las secciones relacionadas incluyen
44 Poner nombre a las cosas
Respetar la cultura
Coherencia
Cambiar nombres es aún más difícil
Las secciones relacionadas incluyen
Retos

8. Antes del proyecto

45 El pozo de los requisitos
El mito de los requisitos
Programar como terapia
Los requisitos son un proceso
Póngase en la piel del cliente
Requisitos frente a política
Requisitos frente a realidad
Documentar requisitos
Sobreespecificación
Solo una cosita más...
Mantenga un glosario
Las secciones relacionadas incluyen
Ejercicios
Retos
46 Resolver rompecabezas imposibles
Grados de libertad
¡No se estorbe a sí mismo!
La suerte favorece a la mente preparada
Las secciones relacionadas incluyen
Retos
47 Trabajar juntos
Programación en pareja
Programación en grupo
¿Qué debería hacer?
48 La esencia de la agilidad
Nunca puede haber un proceso ágil
Entonces, ¿qué hacemos?
Y esto dirige el diseño
Las secciones relacionadas incluyen
Retos

9. Proyectos pragmáticos

49 Equipos pragmáticos
Sin ventanas rotas

Ranas hervidas
Planifique su cartera de conocimientos
Comunique la presencia del equipo
No se repitan
Balas trazadoras en equipos
Automatización
Saber cuándo dejar de añadir pintura
Las secciones relacionadas incluyen
Retos
50 Los cocos no sirven
El contexto importa
La talla única no le queda bien a nadie
El objetivo real
Las secciones relacionadas incluyen
51 Kit pragmático básico
Dirija con el control de versiones
Pruebas despiadadas y continuas
Reforzar la red
Automatización completa
Las secciones relacionadas incluyen
Retos
52 Deleite a sus usuarios
Las secciones relacionadas incluyen
53 Orgullo y prejuicio

Posfacio

La brújula moral
Imagine el futuro que quiere

Bibliografía

Posibles respuestas a los ejercicios

Créditos

Prólogo

Recuerdo cuando Dave y Andy tuitearon por primera vez acerca de la nueva edición de este libro. Fue una gran noticia. Vi como la comunidad de creadores de código respondía con entusiasmo. Mi página de inicio bullía con expectación. Después de más de veinte años, *El programador pragmático* es tan relevante hoy como lo era entonces.

Es muy significativo que un libro con tanta historia generase semejante reacción. Tuve el privilegio de leer una copia inédita para escribir este prólogo, y entendí por qué había causado tanto alboroto. Aunque se trata de un libro técnico, no es justo llamarlo así. Los libros técnicos a menudo intimidan. Están repletos de palabras imponentes, términos confusos, ejemplos enrevesados que, sin querer, hacen que te sientas estúpido. Cuanto más experimentado es el autor, más fácil es olvidar cómo es aprender conceptos nuevos, ser un principiante.

A pesar de sus décadas de experiencia en programación, Dave y Andy han superado el difícil reto de escribir con el entusiasmo de personas que acaban de aprender estas lecciones. No nos hablan con condescendencia. No asumen que somos expertos. Ni siquiera dan por hecho que hemos leído la primera edición. Nos aceptan como somos, programadores que quieren ser mejores, sin más. Dedican las páginas de este libro a ayudarnos a conseguirlo, paso factible a paso factible.

Para ser justos, ya han hecho esto antes. La publicación original estaba llena de ejemplos tangibles, ideas nuevas y consejos prácticos para ejercitar nuestros músculos y desarrollar nuestro cerebro para la escritura de código que todavía se aplican hoy en día. Pero esta actualización realiza dos mejoras en el libro.

La primera es la más evidente: elimina algunas de las referencias antiguas y los ejemplos desfasados y los sustituye por contenido reciente y moderno. No encontrará ejemplos de ciclos invariantes ni *build machines*. Dave y Andy han recogido su contenido potente y se han asegurado de que

las lecciones todavía se entienden, sin las distracciones de los ejemplos antiguos. Desempolvan viejas ideas, como DRY (*don't repeat yourself*, no te repitas) y le dan una mano de pintura fresca, haciendo que brillen.

La segunda, sin embargo, es la que hace este lanzamiento realmente emocionante. Después de escribir la primera edición, tuvieron la oportunidad de reflexionar sobre lo que estaban intentando decir, con lo que querían que se quedasen los lectores y cómo estaba percibiéndose. Recibieron *feedback* sobre esas lecciones. Vieron qué calaba, qué había que perfeccionar, qué se entendía mal. En los veinte años en los que este libro ha pasado por las manos y los corazones de programadores de todo el mundo, Dave y Andy han estudiado esta respuesta y han formulado nuevas ideas, nuevos conceptos.

Han aprendido la importancia de la capacidad de acción y han reconocido que podría decirse que los desarrolladores tienen más capacidad de acción que muchos otros profesionales. Empiezan este libro con un mensaje sencillo, pero profundo: "Es su vida". Nos recuerda nuestro propio poder en nuestra base de código, en nuestros empleos, en nuestras carreras. Marca el tono del resto del libro; es más que otro libro técnico lleno de ejemplos de código.

Lo que hace que destaque de verdad en las estanterías de los libros técnicos es que entiende lo que significa ser programadores. La programación trata sobre intentar hacer el futuro menos doloroso, sobre hacer cosas mal y ser capaz de recuperarse, sobre crear buenos hábitos, sobre entender tu conjunto de herramientas. La creación de código es solo una parte del mundo de los programadores, y este libro explora ese mundo.

Paso mucho tiempo pensando en el viaje de la creación de código. Yo no crecí creando código. No lo estudié en la universidad. No me pasé la adolescencia jugueteando con la tecnología. Entré en el mundo del código a los veintitantos y tuve que aprender lo que significaba ser programadora. Esta comunidad es muy diferente de otras de las que he formado parte. Hay una dedicación única al aprendizaje y la practicidad es al mismo tiempo refrescante e intimidante.

Para mí, fue como entrar en un mundo nuevo. O, al menos, una nueva ciudad. Tuve que conocer a los vecinos, elegir un supermercado, encontrar las mejores cafeterías. Me llevó un tiempo familiarizarme con el terreno, encontrar las rutas más eficientes, evitar las calles con más tráfico, saber

cuándo era probable que hubiese atascos. El clima era diferente, necesitaba un fondo de armario nuevo.

Las primeras semanas, incluso los primeros meses, en una ciudad nueva, pueden ser aterradores. ¿No sería maravilloso tener un vecino amable y bien informado que lleve un tiempo viviendo allí? ¿Alguien que pueda enseñarte el lugar, decirte cuáles son esas cafeterías? ¿Una persona que lleve ahí el tiempo suficiente para conocer la cultura, entender el ritmo de la ciudad, de forma que no solo ayude a que te sientas en casa, sino también a que te conviertas en un miembro que contribuye a la comunidad? Dave y Andy son esos vecinos.

Como recién llegada relativa, es fácil sentirse desbordada, no por el acto de programar, sino por el proceso de convertirse en programadora. Es necesario que se produzca un cambio de mentalidad completo, un cambio en los hábitos, los comportamientos y las expectativas. El proceso de conversión en mejores programadores no ocurre solo porque sepamos escribir código; debe haber también intención y práctica deliberada. Este libro es una guía para convertirse en programadores mejores de manera eficiente.

Pero no se equivoque: no le dice cómo debería ser la programación. No es filosófico ni sentencioso. Dice, de manera llana y simple, qué es un programador pragmático, cómo trabaja, cómo enfoca el código. Dejan a su elección si quiere ser uno. Si siente que eso no le va, no le guardan rencor. Pero, si decide que le interesa, son los vecinos amables que están ahí para mostrarle el camino.

—Saron Yitbarek.
Fundadora y directora ejecutiva de CodeNewbie.
Presentadora de *Command Line Heroes.*

Prefacio de la segunda edición

En los años noventa, trabajábamos con empresas cuyos proyectos estaban teniendo problemas. Nos dimos cuenta de que estábamos diciendo lo mismo todo el tiempo: quizá debería probar eso antes de enviarlo; ¿por qué el código solo se construye en el ordenador de Mary? ¿Por qué nadie ha preguntado a los usuarios?

Para ahorrar tiempo con los nuevos clientes, empezamos a tomar notas. Y esas notas se convirtieron en *El programador pragmático*. Para nuestra sorpresa, pareció calar hondo, y ha seguido siendo popular durante los últimos 20 años.

Pero más de dos décadas son varias vidas cuando hablamos de software. Traiga a un desarrollador de 1999 y métalo en un equipo actual, y verá los apuros que pasa en este mundo nuevo y extraño. Pero el mundo de los noventa es igual de raro para los desarrolladores de hoy. Las referencias de este libro a cosas como CORBA, herramientas CASE y bucles indexados eran como mucho pintorescas y, probablemente, confusas.

Al mismo tiempo, 20 años no han afectado para nada al sentido común. La tecnología ha cambiado, pero las personas no. Las prácticas y los enfoques que eran buena idea entonces siguen siéndolo ahora. Esos aspectos del libro han envejecido bien.

Así pues, cuando llegó la hora de crear esta edición por el 20º aniversario, tuvimos que tomar una decisión. Podíamos repasar y actualizar las tecnologías a las que hacemos referencia y santas pascuas, o podíamos reexaminar los supuestos que había detrás de las prácticas que recomendábamos a la luz de las dos décadas de experiencia adicionales.

Al final, hicimos las dos cosas.

Como resultado, este libro es una especie de paradoja del barco de Teseo.[1] Alrededor de un tercio de los temas del libro son completamente nuevos. Del resto, la mayoría se ha reescrito, de manera parcial o total.

Nuestra intención era hacer que las cosas fuesen más claras, más relevantes y, de alguna manera, atemporales.

Hemos tomado decisiones difíciles. Hemos desechado el apéndice "Recursos", porque sería imposible mantenerlo actualizado y porque es más fácil que busque lo que quiera. Hemos reorganizado y reescrito temas relacionados con la concurrencia, dada la abundancia actual de hardware paralelo y la escasez de buenas maneras de tratar con él. Hemos añadido contenido que refleja actitudes y entornos cambiantes, desde el movimiento del desarrollo ágil que ayudamos a iniciar hasta la aceptación cada vez mayor de modismos en la programación funcional y la necesidad creciente de tener en cuenta la privacidad y la seguridad. Sin embargo, resulta interesante que hubiese bastante menos discusión entre nosotros sobre el contenido de esta edición que cuando escribimos la primera. Los dos sentíamos que todo aquello que era importante era más fácil de identificar. En cualquier caso, este libro es el resultado. Por favor, disfrútelo. Puede adoptar, quizá, algunas prácticas nuevas. Tal vez decida que algunas cosas de las que sugerimos están mal. Involúcrese en su oficio. Denos su opinión.

Pero, sobre todo, recuerde divertirse con esto.

Cómo se organiza el libro

Este libro está escrito como una colección de temas cortos. Cada tema es independiente y gira en torno a una idea central específica. Encontrará numerosas referencias cruzadas, que le ayudarán a situar cada tema dentro de un contexto. Puede leer los temas en cualquier orden, no es necesario que los lea tal y como aparecen.

De manera ocasional, se encontrará con un cuadro titulado Truco nn (como Truco 1, "Preocúpese por su oficio"). Además de enfatizar puntos del texto, creemos que los trucos tienen vida propia, influyen en nuestra vida a diario.

Hemos incluido ejercicios y retos donde es pertinente. Los ejercicios suelen tener respuestas relativamente directas, mientras que los retos son más abiertos. Para que se haga una idea de nuestra forma de pensar, hemos incluido nuestras respuestas a los ejercicios al final del libro, pero muy pocos tienen una única solución correcta. Los retos pueden servir como

base para debates o para un ensayo acerca de los cursos de programación avanzados. También hay una breve bibliografía que incluye los libros y artículos a los que hacemos referencias explícitas.

¿Qué hay en un nombre?

> *—Cuando yo uso una palabra —insistió Humpty Dumpty con un tono de voz más bien desdeñoso — quiere decir lo que yo quiero que diga..., ni más ni menos.*
>
> *—Lewis Carroll, A través del espejo.*

Esparcidos por el libro encontrará varios ejemplos de jerga, bien palabras perfectamente correctas en nuestro idioma que se han corrompido para que signifiquen algo técnico, bien palabras inventadas horribles a las que han asignado significados algunos científicos de la computación que se la tienen jurada a la lengua. La primera vez que usemos alguna de estas palabras de jerga, intentaremos definirla o, al menos, dar alguna pista sobre su significado. Sin embargo, estamos seguros de que algunas han pasado inadvertidas y otras, como “objeto” y “base de datos relacional”, tienen un uso bastante común que hace que añadir una definición sea aburrido. Si se encuentra con un término que no ha visto antes, por favor, no lo ignore sin más. Tómese un tiempo para buscarlo, quizá en la web o tal vez en un libro de texto de ciencias de la computación. Y, si tiene ocasión, escríbanos un correo electrónico y quéjese para que añadamos una definición en la siguiente edición.

Dicho esto, hemos decidido vengarnos de los científicos de la computación. A veces, hay palabras de jerga perfectamente válidas para conceptos, palabras que hemos decidido ignorar. ¿Por qué? Porque la jerga existente suele restringirse al dominio de un problema particular o una fase concreta del desarrollo. Sin embargo, una de las filosofías básicas de este libro es que la mayoría de las técnicas que recomendamos son universales: la modularidad se aplica al código, los diseños, la documentación y la organización de los equipos, por ejemplo. Cuando queríamos usar la palabra de jerga convencional en un contexto más amplio, resultaba confuso; parecía que no podíamos deshacernos del bagaje que llevaba consigo el

término original. Cuando ha ocurrido eso, hemos contribuido al declive de la lengua inventando nuestros propios términos.

Código fuente y otros recursos

La mayor parte del código que se muestra en este libro se ha extraído de archivos fuente compilables, que se encuentran disponibles en la página web de Anaya Multimedia (http://www.anayamultimedia.es). Vaya al botón Selecciona Complemento de la ficha del libro, donde podrá descargar el código fuente.

También dispone de estos archivos en la página web del libro original en https://pragprog.com/titles/tpp20. Ahí también encontrará enlaces a recursos útiles, además de actualizaciones del libro original y otras noticias en inglés.

---

[1] Si a lo largo de los años se sustituyen todos los componentes de un barco, ¿la nave resultante es el mismo barco?

Del prefacio de la primera edición

Este libro le ayudará a convertirse en un desarrollador mejor.

Puede que sea un desarrollador solitario, un miembro de un equipo grande para proyectos o un asesor que trabaja con muchos clientes a la vez. No importa; este libro le ayudará, como individuo, a hacer un trabajo mejor. No se trata de un libro teórico; nos centramos en temas prácticos, en que utilice su experiencia para tomar decisiones mejor fundamentadas. La palabra "pragmático" viene del latín *pragmaticus* ("hábil en asuntos"), que a su vez deriva del griego πραγματικός, que significa "adecuado para el uso".

Este libro va de "hacer".

La programación es un oficio artesano. En su forma más simple, se reduce a conseguir que un ordenador haga lo que queremos que haga (o lo que nuestros usuarios quieran que haga). Como programador, es en parte oyente, en parte consejero, en parte intérprete y en parte dictador. Intenta capturar requisitos imprecisos y encontrar una manera de expresarlos de forma que una simple máquina pueda hacerles justicia. Intenta documentar su trabajo para que otros puedan entenderlo, e intenta diseñar su trabajo de forma que otras personas pueda construir sobre él. Es más, trata de hacer todo eso mientras se enfrenta al paso inexorable del tiempo marcado para un proyecto. Hace pequeños milagros todos los días.

Es un trabajo difícil.

Hay mucha gente que le ofrece ayuda. Distribuidores de herramientas promocionan los milagros que realizan sus productos. Gurús de las metodologías prometen que sus técnicas garantizan resultados. Todo el mundo afirma que su lenguaje de programación es el mejor y que cada sistema operativo es la respuesta a todos los males concebibles.

Por supuesto, nada de esto es cierto. No hay respuestas fáciles. No hay una "mejor solución", ya sea una herramienta, un lenguaje o un sistema

operativo. Solo puede haber sistemas que son más apropiados en un conjunto de circunstancias determinado.

Aquí es donde entra el pragmatismo. No debería aferrarse a una tecnología concreta, sino tener una base de fondo y experiencia lo bastante amplia para que le permita elegir soluciones buenas en situaciones particulares. Ese fondo surge de un entendimiento de los principios básicos de la ciencia de la computación, y su experiencia viene de una amplia gama de proyectos prácticos. La teoría y la práctica se combinan para hacerle más fuerte.

Debe ajustar su enfoque para adaptarse a las circunstancias y el entorno actuales. Juzgue la importancia relativa de todos los factores que afecten a un proyecto y use su experiencia para generar las soluciones apropiadas. Haga esto de manera continua a medida que progresa el trabajo. Los programadores pragmáticos acaban el trabajo, y lo hacen bien.

¿Quién debería leer este libro?

Este libro va dirigido a quienes quieran ser programadores más efectivos y más productivos. Quizá se sienta frustrado porque le parece que no está aprovechando su potencial. Puede que se fije en colegas que parecen usar herramientas que les hacen más productivos que usted. Tal vez su trabajo actual utiliza tecnologías antiguas y quiere saber cómo pueden aplicarse ideas más nuevas a lo que hace.

No fingimos tener todas (ni siquiera la mayoría de) las respuestas, ni todas nuestras ideas son aplicables en todas las situaciones. Lo único que podemos decir es que, si sigue nuestro enfoque, ganará experiencia con rapidez, aumentará su productividad y tendrá un mejor entendimiento del proceso completo de desarrollo. Y escribirá software mejor.

¿Qué hace a un programador pragmático?

Cada programador es único, con puntos fuertes y débiles, preferencias y antipatías individuales. Con el tiempo, cada uno elabora su propio entorno personal. Ese entorno reflejará la individualidad del programador con tanta

claridad como sus aficiones, su ropa o su corte de pelo. Sin embargo, si es un programador pragmático, compartirá muchas de las siguientes características:

- Primero en adoptar/se adapta con rapidez.
  Tiene buen instinto para las tecnologías y las técnicas y le encanta probar cosas. Cuando le dan algo nuevo, enseguida le coge el tranquillo y lo integra con el resto de sus conocimientos. Su confianza nace de la experiencia.
- Inquisitivo.
  Tiende a hacer preguntas. “Eso está muy bien, ¿cómo lo has hecho? ¿Tienes problemas con esa biblioteca? ¿Qué es esa informática cuántica de la que he oído hablar? ¿Cómo se implementan los enlaces simbólicos?” Es un coleccionista de hechos, cada uno de los cuales puede afectar a sus decisiones dentro de unos años.
- Pensador crítico.
  Rara vez da las cosas por sentadas sin estudiar primero los hechos. Cuando un colega dice: “Porque así es como se hace” o un distribuidor promete la solución a todos sus problemas, huele un desafío.
- Realista.
  Intenta entender la naturaleza subyacente de cada problema al que se enfrenta. Este realismo le da una impresión bastante exacta de lo difíciles que son las cosas y cuánto tiempo llevarán. Una comprensión profunda de que un proceso debería ser difícil o tardará bastante en completarse le proporciona la resistencia para seguir trabajando en ello.
- Versátil.
  Se esfuerza por familiarizarse con una amplia variedad de tecnologías y entornos y trabaja para mantenerse al día con los nuevos desarrollos. Aunque puede que su trabajo actual requiera que sea un especialista, siempre será capaz de pasar a nuevas áreas y nuevos retos.

Hemos dejado las características más básicas para el final. Todos los programadores pragmáticos las comparten y son lo bastante básicas para

expresarse como trucos:

> **Truco 1.** Preocúpese por su oficio.

Nos parece que no tiene sentido desarrollar software a menos que nos preocupemos por hacerlo bien.

> **Truco 2.** ¡Piense! En su trabajo.

Para ser un programador pragmático, le desafiamos a pensar en lo que está haciendo mientras lo está haciendo. No se trata de una inspección única de prácticas actuales, sino de una valoración continua de cada decisión que toma, cada día y en cada proyecto. No avance nunca con el piloto automático puesto. Debe pensar constantemente y ser crítico con su trabajo en tiempo real. El antiguo eslogan corporativo de IBM, *THINK!* (¡PIENSA!), es el mantra del programador pragmático.

Si le parece que esto es difícil para usted, entonces está mostrando el rasgo realista. Esto le va a llevar algo de tiempo valioso, tiempo que es probable que ya esté bajo una presión enorme. La recompensa es una implicación más activa en un trabajo que le gusta, una sensación de dominio de una variedad cada vez mayor de temas y el placer de un sentimiento de mejora continua. A largo plazo, la inversión de tiempo dará sus frutos cuando usted y su equipo se vuelvan más eficientes, escriban código más fácil de mantener y pasen menos tiempo en reuniones.

Pragmatistas individuales, equipos grandes

Algunas personas sienten que no hay sitio para la individualidad en equipos grandes o proyectos complejos. Dicen que el software es una disciplina de ingeniería que se descompone si los miembros individuales del equipo empiezan a tomar decisiones por su cuenta.

No estamos de acuerdo en absoluto.

Debería haber ingeniería en la creación del software. Sin embargo, eso no excluye la artesanía individual. Piense en las grandes catedrales construidas en Europa durante la Edad Media. Cada una de ellas necesitó miles de años-hombre de esfuerzo, distribuidos a lo largo de muchas décadas. Las lecciones aprendidas se transmitían a la siguiente generación de constructores, que hacían avanzar el estado de la ingeniería estructural con sus logros. Pero los carpinteros, los canteros, los tallistas y los vidrieros eran artesanos, e interpretaban los requisitos de ingeniería para producir un conjunto que trascendía el aspecto puramente mecánico de la construcción. Su fe en sus contribuciones individuales era lo que sustentaba los proyectos: los que cortamos solo simples piedras debemos visualizar catedrales.

Dentro de la estructura general de un proyecto, siempre hay espacio para la individualidad y la artesanía. Esto es cierto en particular si se tiene en cuenta el estado actual de la ingeniería de software. Dentro de cien años, puede que nuestra ingeniería parezca tan arcaica como las técnicas usadas por los constructores de catedrales medievales pueden parecer a los ingenieros actuales, mientras que se seguirá honrando nuestra artesanía.

Es un proceso continuo

Un turista que visitaba el Eton College en Inglaterra preguntó al jardinero cómo lograba que el césped fuese tan perfecto.
—Es fácil. Solo hay que retirar el rocío todas las mañanas, cortar cada dos días y pasar el rodillo una vez a la semana —respondió el jardinero.
—¿Eso es todo? —preguntó el turista.
—Desde luego —respondió el jardinero—. Hágalo durante 500 años y también tendrá un césped así de bonito.

Los buenos céspedes necesitan pequeñas cantidades de cuidados diarios, y lo mismo ocurre con los buenos programadores. A los consultores de gestión les gusta meter la palabra *kaizen* en las conversaciones. “*Kaizen*” es un término japonés que captura el concepto de hacer muchas mejoras pequeñas de manera continua. Se consideró que fue una de las principales razones para la espectacular ganancia en productividad y calidad en la fabricación japonesa y se copió por todo el mundo. El *kaizen* se aplica también a los individuos. Trabaje cada día para perfeccionar sus habilidades

y para añadir nuevas herramientas a su repertorio. A diferencia del césped de Eton, empezará a ver resultados en cuestión de días. Con los años, le impresionará ver cuánto ha florecido su experiencia y cuánto han crecido sus habilidades.

1
Una filosofía pragmática

Este libro es sobre usted.

No se confunda, es su carrera y, lo que es más importante, “Es su vida”. Le pertenece. Está aquí porque sabe que puede convertirse en un desarrollador mejor y ayudar a otros a ser mejores también. Puede convertirse en un programador pragmático.

¿Qué distingue a los programadores pragmáticos? Consideramos que es una actitud, un estilo, una filosofía sobre la forma de abordar los problemas y sus soluciones. Piensan más allá del problema inmediato, lo sitúan en un contexto más grande y buscan el panorama general. Al fin y al cabo, sin ese contexto más grande, ¿cómo podemos ser pragmáticos? ¿Cómo podemos llegar a acuerdos inteligentes y tomar decisiones bien fundadas?

Otra clave para su éxito es que los programadores pragmáticos asumen la responsabilidad de todo lo que hacen, algo que veremos en “El gato se comió mi código fuente”. Al ser responsables, los programadores pragmáticos no van a sentarse tranquilamente a ver cómo sus proyectos se vienen abajo por negligencias. En “Entropía del software”, le explicamos cómo mantener sus proyectos impolutos. A la mayoría de la gente le resulta difícil el cambio, unas veces por buenas razones y otras por una simple cuestión de inercia. En “Sopa de piedras y ranas hervidas”, echamos un vistazo a una estrategia para instigar el cambio y (por el interés del equilibrio) presentamos el cuento admonitorio de un anfibio que ignoraba los peligros del cambio gradual.

Uno de los beneficios de entender el contexto en el que trabajamos es que se vuelve más fácil saber lo bueno que tiene que ser nuestro software. A veces, “casi perfecto” es la única opción posible, pero, a menudo, hay que sacrificar algo a cambio. Exploramos esto en “Software lo bastante bueno”.

Por supuesto, necesita tener una amplia base de conocimiento y experiencia para sacar todo esto adelante. El aprendizaje es un proceso

continuo y en desarrollo. En “Su cartera de conocimientos”, vemos algunas estrategias para mantener el ímpetu.

Por último, ninguno de nosotros trabaja en el vacío. Todos pasamos gran parte del tiempo interactuando con otras personas. “¡Comuníquese!” muestra una lista de maneras en que podemos hacer esto mejor. La programación pragmática surge de una filosofía de pensamiento pragmático. Este capítulo determina la base para esa filosofía.

1 Es su vida

> *No estoy en este mundo para cumplir tus expectativas, y tú no estás en este mundo para cumplir las mías.*
>
> *—Bruce Lee.*

Es su vida. Le pertenece. Usted la dirige. Usted la crea.

Muchos desarrolladores con los que hablamos se sienten frustrados. Sus preocupaciones son variadas. Algunos sienten que están estancándose en su trabajo, otros que la tecnología ha pasado de largo a su lado. Hay personas que sienten que están infravaloradas, que están mal pagadas o que sus equipos son tóxicos. A lo mejor quieren mudarse a Asia o Europa, o trabajar desde casa.

Y la respuesta que les damos es siempre la misma.

“¿Por qué no puedes cambiarlo?”.

El desarrollo de software debe aparecer cerca de lo más alto en cualquier lista de carreras en las que tenemos control. Hay demanda para nuestras habilidades, nuestros conocimientos cruzan fronteras geográficas, podemos trabajar de manera remota. Nos pagan bien. Realmente podemos hacer lo que queramos.

Pero, por alguna razón, los desarrolladores parecen resistirse al cambio. Se apoltronan y esperan que las cosas mejoren. Se quedan mirando con pasividad mientras sus habilidades se quedan anticuadas y se quejan de que sus empresas no les ofrecen formación. Se fijan en los anuncios de destinos exóticos mientras van en el autobús y, después, salen a la lluvia heladora y caminan con pesadez hasta el trabajo.

Así pues, aquí está el truco más importante del libro.

Truco 3. Tiene capacidad de acción.

¿Su entorno laboral es un asco? ¿Le aburre su trabajo? Intente arreglarlo, pero no se pase la vida intentándolo. Como dice Martin Fowler, "puede cambiar su organización o cambiar su organización".[1]

Si le parece que la tecnología está pasando de largo respecto a usted, saque tiempo (a su propio ritmo) para estudiar cosas nuevas que parezcan interesantes. Está invirtiendo en usted mismo, así que es razonable que lo haga fuera de su horario laboral.

¿Quiere trabajar de manera remota? ¿Lo ha pedido? Si le dicen que no, busque a alguien que diga que sí.

Esta industria ofrece un conjunto notable de oportunidades. Sea proactivo y aprovéchelas.

Las secciones relacionadas incluyen

- Tema 4, "Sopa de piedras y ranas hervidas".
- Tema 6, "Su cartera de conocimientos".

2 El gato se comió mi código fuente

La mayor debilidad de todas es el miedo a parecer débil.

—J.B. Bossuet, Política deducida de las palabras propias de la Sagrada Escritura, 1709.

Una de las piedras angulares de la filosofía pragmática es la idea de asumir la responsabilidad por nosotros mismos y nuestras acciones en lo que respecta al avance de nuestra carrera, nuestro aprendizaje y nuestra educación, nuestro proyecto y nuestro trabajo diario. Los programadores pragmáticos se hacen cargo de su propia carrera y no tienen miedo de admitir el desconocimiento o el error. No es el aspecto más agradable de la programación, desde luego, pero ocurrirá, incluso en los mejores proyectos. Pese a las pruebas exhaustivas, la buena documentación y la automatización sólida, hay cosas que salen mal. Las entregas se retrasan. Surgen problemas técnicos imprevistos.

Estas cosas pasan e intentamos lidiar con ellas de la forma más profesional posible. Eso significa ser honestos y directos. Podemos estar orgullosos de nuestras habilidades, pero debemos asumir la responsabilidad sobre nuestros defectos, nuestra ignorancia y nuestros errores.

Confianza del equipo

Por encima de todo, su equipo necesita poder confiar en y depender de usted, y usted necesita sentirse cómodo dependiendo de cada uno de ellos también. Confiar en un equipo es esencial para la creatividad y la colaboración, según muestran diversos trabajos de investigación.[2] En un entorno sano basado en la confianza, podemos decir lo que pensamos de forma segura, presentar nuestras ideas y depender de los miembros del equipo, que a su vez pueden depender de nosotros. Si no hay confianza, bueno...

Imagine un equipo de ninjas sigilosos con alta tecnología que se infiltra en la guarida de un villano. Tras meses de planificación y perfecta ejecución, han llegado al lugar. Ahora, es su turno para activar la cuadrícula de guía láser: “Lo siento, chicos, no tengo el láser. El gato estaba jugando con el puntito rojo y lo he dejado en casa”.

Ese tipo de traición de la confianza debe ser difícil de arreglar.

Asuma la responsabilidad

La responsabilidad es algo a lo que accedemos de manera activa. Adquirimos un compromiso para garantizar que algo se hace bien, pero tenemos necesariamente un control directo sobre cada aspecto de ello. Además de dar lo mejor de nosotros mismos, debemos analizar la situación para detectar riesgos que quedan fuera de nuestro control. Tenemos derecho a no asumir una responsabilidad sobre una situación imposible, o una en la que los riesgos son demasiado grandes, o las implicaciones éticas son demasiado sospechosas. Tendrá que tomar la decisión en función de sus propios valores y su criterio.

Cuando acepte la responsabilidad sobre un resultado, debería esperar tener que responder por él. Cuando cometa una equivocación (como

hacemos todos) o un error de juicio, admítalo con honestidad e intente proponer opciones.

No culpe a otra persona ni circunstancia, ni se invente excusas. No eche la culpa a un distribuidor, a un lenguaje de programación, al director o a sus compañeros. Puede que todas esas cosas hayan influido, pero depende de usted ofrecer soluciones, no excusas.

Si existía un riesgo de que el distribuidor le fallase, debería haber tenido un plan de contingencia. Si su unidad de almacenamiento masivo se estropea (y con ella se pierde todo su código fuente) y no tiene una copia de seguridad, es culpa suya. Decirle a su jefe "el gato se ha comido mi código fuente" no servirá.

Truco 4. Ofrezca opciones, no se invente excusas pobres.

Antes de acercarse a alguien a decirle por qué algo no puede hacerse, va con retraso o está estropeado, pare y escúchese. Hable con el patito de goma de su monitor o con el gato. ¿Su excusa suena razonable o estúpida? ¿Cómo va a sonarle a su jefe?

Repase la conversación en su mente. ¿Qué es probable que diga la otra persona? ¿Le preguntará si ha probado esto o ha considerado aquello? ¿Cómo responderá usted? Antes de ir a darle a esa persona la mala noticia, ¿hay algo más que pueda probar? A veces, ya sabe lo que van a decirle, así que ahórreles la molestia.

En vez de excusas, ofrezca opciones. No diga que no puede hacerse; explique qué puede hacerse para resolver la situación. ¿Hay que eliminar código? Dígaselo y explique el valor de la refactorización (véase el tema 40, "Refactorización").

¿Necesita dedicar tiempo a crear un prototipo para determinar la mejor manera de proceder (véase el tema 13, "Prototipos y notas en *post-its*")? ¿Tiene que introducir pruebas mejores (véase el tema 41, "Probar para escribir código" y "Pruebas despiadadas y continuas" en el capítulo 9) o automatización para evitar que vuelva a ocurrir?

Quizá necesite recursos adicionales para completar esta tarea. ¿O tal vez necesite pasar más tiempo con los usuarios? Quizá la cuestión sea usted mismo: ¿necesita aprender alguna técnica o tecnología en mayor

profundidad? ¿Le ayudaría algún libro o curso? No tenga miedo de pedir o de admitir que necesita ayuda.

Intente olvidar las malas excusas antes de decirlas en voz alta. Si no le queda más remedio que usarlas, dígaselo primero a su gato. Al fin y al cabo, si el pequeño Calcetines va a cargar con la culpa...

Las secciones relacionadas incluyen

- Tema 49, “Equipos pragmáticos”.

Retos

- ¿Cómo reacciona cuando alguien (el cajero del banco, el mecánico del taller o un dependiente) le pone una mala excusa? ¿Qué piensa de esa persona y de su empresa como resultado?
- Cuando se oiga a sí mismo decir: “No lo sé”, asegúrese de añadir después: “Pero lo averiguaré”. Es una manera genial de admitir lo que no sabe, pero asumiendo después la responsabilidad como un profesional.

3 Entropía del software

Aunque el desarrollo de software es inmune a casi todas las leyes de la física, el inexorable incremento de la entropía nos afecta de pleno. La entropía es un término de física que se refiere a la cantidad de “desorden” en un sistema. Por desgracia, las leyes de la termodinámica garantizan que la entropía en el universo tienda hacia el máximo. Cuando aumenta el desorden en el software, lo denominamos “pudrición del software”. Algunas personas prefieren utilizar el término más optimista “deuda técnica”, con la noción implícita de que la pagarán algún día. Es probable que no lo hagan.

En cualquier caso, al margen del nombre, tanto la deuda como la pudrición pueden extenderse sin control.

Existen muchos factores que pueden contribuir a la pudrición del software. El más importante parece ser la psicología, o la cultura, que se emplea en un proyecto. Incluso si somos un equipo de uno, la psicología de nuestro proyecto puede ser una cuestión muy delicada. A pesar de la planificación escrupulosa y las personas más adecuadas, un proyecto puede deteriorarse y arruinarse durante su vida. Y, por otra parte, hay otros proyectos que, pese a las enormes dificultades y los contratiempos constantes, luchan con éxito contra la tendencia natural hacia el desorden y consiguen llegar a buen puerto.

¿Qué marca la diferencia?

En el centro de las ciudades, algunos edificios son bonitos y están limpios, mientras que otros son armatostes deteriorados. ¿Por qué? Los investigadores en el campo del crimen y el deterioro urbano descubrieron un mecanismo desencadenante fascinante, uno que convierte con rapidez un edificio limpio, intacto y habitado en uno ruinoso y abandonado.[3]

Una ventana rota.

Una ventana rota que se deje sin reparar durante un periodo de tiempo considerable infunde en los habitantes del edificio una sensación de abandono, una sensación de que a los responsables no les importa el edificio. Así que se rompe otra ventana. La gente empieza a tirar basura. Aparecen grafitis. Comienza a producirse un daño estructural serio. En un periodo de tiempo relativamente corto, el edificio queda demasiado dañado para que el propietario quiera arreglarlo, y la sensación de abandono se convierte en una realidad.

¿Qué marcaría una diferencia? Varios psicólogos han realizado estudios[4] que demuestran que la desesperanza puede ser contagiosa. Piense en el virus de la gripe en espacios cerrados. Ignorar una situación que está claro que va mal refuerza la idea de que quizá nada puede arreglarse, que no le importa a nadie, que todo está abocado al fracaso; son todo pensamientos negativos que se extienden por los miembros del equipo, lo que genera un círculo vicioso.

Truco 5. No viva con ventanas rotas.

No deje “ventanas rotas” (malos diseños, decisiones equivocadas o código pobre) sin reparar. Arregle cada una en cuanto la descubra. Si no hay tiempo suficiente para arreglarla de forma adecuada, tápela con tablas de manera temporal. Quizá pueda comentar el código irritante, mostrar un mensaje que diga “Sin implementar” o sustituir con datos *dummy*. Haga algo para evitar más daños y para demostrar que está por encima de la situación.

Hemos visto sistemas limpios y funcionales que se deterioran con bastante rapidez una vez que las ventanas empiezan a romperse. Hay otros factores que pueden contribuir a la pudrición del software, los veremos más adelante, pero la negligencia acelera la pudrición más que ningún otro factor.

Puede que esté pensando que nadie tiene tiempo de andar limpiando todos los cristales rotos de un proyecto. Si es así, más vale que empiece a planear hacerse con un contenedor de basura o mudarse a otro vecindario. No deje que la entropía gane.

Primero, no haga daño

Andy tenía un conocido que era obscenamente rico. Su casa estaba inmaculada, llena de antigüedades de valor incalculable, obras de arte tridimensionales y cosas así. Un día, un tapiz que estaba colgado demasiado cerca de una chimenea empezó a arder. Los bomberos llegaron corriendo para salvar la situación y la casa. Pero, antes de arrastrar las mangueras grandes y sucias por el interior, se detuvieron (con las llamas propagándose) para extender una alfombra entre la puerta principal y el origen del incendio.

No querían estropear la moqueta.

Sí, suena bastante exagerado. Desde luego, la máxima prioridad de los bomberos es sofocar el incendio, y al cuerno con los daños colaterales. Pero estaba claro que habían evaluado la situación, confiaban en su capacidad para controlar el fuego y tuvieron cuidado de no infligir un daño innecesario a la propiedad. Eso es lo que debe hacerse con el software: no causar daños colaterales solo porque hay algún tipo de crisis. Una ventana rota ya son demasiadas.

Una ventana rota (una porción de código mal diseñada, una mala decisión de la dirección con la que el equipo debe vivir mientras dure el proyecto) es lo único que hace falta para comenzar el declive. Si se da cuenta de que está trabajando en un proyecto con bastantes ventanas rotas, es demasiado fácil adoptar la mentalidad de que "todo el resto de este código es una porquería, voy a seguir en esa línea, sin más". No importa si el proyecto ha ido bien hasta ese punto. En el experimento original que llevó a la teoría de las ventanas rotas, un coche abandonado se quedó sin tocar durante una semana. Pero, en cuanto se rompió una sola ventana, el coche fue desmantelado en cuestión de horas.

Del mismo modo, si se encuentra trabajando en un proyecto en el que el código es bonito y prístino (escrito con limpieza, bien diseñado y elegante), es probable que ponga especial cuidado en no estropearlo, igual que los bomberos. Incluso aunque haya un incendio (fecha límite, fecha de lanzamiento, demostración para una feria comercial, etc.), no querrá ser el primero que haga un lío y cause un daño adicional.

Solo dígase: "Nada de ventanas rotas".

Las secciones relacionadas incluyen

- Tema 10, "Ortogonalidad".
- Tema 40, "Refactorización".
- Tema 44, "Poner nombre a las cosas".

Retos

- Ayude a fortalecer a su equipo haciendo un sondeo en el vecindario de su proyecto. Elija dos o tres ventanas rotas y hable con sus compañeros de cuáles son los problemas y qué podría hacerse para arreglarlos.
- ¿Se da cuenta cuando se rompe una ventana por primera vez? ¿Cuál es su reacción? Si ha sido a causa de la decisión de otra persona o de una orden del director, ¿qué puede hacer usted al respecto?

4 Sopa de piedras y ranas hervidas

Tres soldados volvían a casa hambrientos tras la guerra. Cuando vieron un pueblo en la lejanía, se les levantó el ánimo; estaban seguros de que los aldeanos les ofrecerían una comida. Sin embargo, al llegar allí, se encontraron las puertas y las ventanas cerradas. Después de muchos años de guerra, los alimentos escaseaban y los aldeanos hacían acopio de lo que tenían.
Lejos de desistir, los soldados pusieron a hervir una cazuela con agua y metieron con cuidado tres piedras. Los asombrados aldeanos se acercaron a mirar.
—Esto es sopa de piedras —explicaron los soldados.
—¿Eso es todo lo que lleva? —preguntaron los aldeanos.
—¡Desde luego! Aunque algunos dicen que sabe incluso mejor con unas zanahorias...
Un aldeano se fue corriendo y volvió enseguida con una cesta de zanahorias de su reserva.
Unos minutos después, los aldeanos volvieron a preguntar:
—¿Eso es todo?
—Bueno —dijeron los soldados—, un par de patatas le dan cuerpo.
Otro aldeano salió corriendo.
En la siguiente hora, los soldados hicieron una lista con más ingredientes que mejorarían la sopa: carne, puerros, sal y hierbas aromáticas. Cada vez, un aldeano diferente iba corriendo a buscar en su despensa personal.
Al final, tenían una gran cazuela de sopa humeante. Los soldados quitaron las piedras y se sentaron con el resto del pueblo para disfrutar la primera comida decente que cualquiera de ellos había probado en meses.

Hay un par de moralejas en la historia de la sopa de piedras. A los aldeanos los engañan los soldados, que utilizan la curiosidad de los aldeanos para conseguir comida de ellos. Pero lo que es más importante es que los soldados actúan como catalizador, uniendo al pueblo para que todos juntos puedan producir algo que no podrían haber hecho solos, un resultado sinérgico. Al final, todo el mundo gana.

De vez en cuando, quizá le convenga emular a los soldados.

Puede que esté en una situación en la que sepa con exactitud lo que es necesario hacer y cómo hacerlo. El sistema entero aparece ante sus ojos; sabe que está bien. Pero, si pide permiso para abordarlo todo, se encontrará con retrasos y miradas confusas. Se formarán comités, habrá que aprobar presupuestos y las cosas se complicarán. Todo el mundo se guardará sus propios recursos. Esto se denomina a veces “fatiga de arranque”.

Es hora de preparar la sopa de piedras. Determine qué puede pedir de forma razonable. Desarróllelo bien. Una vez que lo tenga, enséñeselo a otras personas y deje que se maravillen. Después, diga: “Por supuesto, estaría mejor si añadiésemos...”. Finja que no es importante. Siéntese y espere a que empiecen a pedirle que añada la funcionalidad que quería

desde el principio. A la gente le resulta más fácil unirse a un éxito en curso. Enséñeles una muestra del futuro y conseguirá su apoyo.[5]

Truco 6. Sea un catalizador para el cambio.

El lado de los aldeanos

Por otra parte, la historia de la sopa de piedras también habla de un engaño sutil y gradual. Tiene que ver con concentrarse demasiado en algo. Los aldeanos piensan en las piedras y se olvidan del resto del mundo. Todos caemos en esa trampa, todos los días. Las cosas nos acechan.

Todos hemos visto los síntomas. Los proyectos van escapándose por completo de nuestro control de forma lenta e inexorable. La mayoría de los desastres de software empiezan siendo demasiado pequeños para que los notemos, y la mayoría de los costes adicionales en los proyectos se producen día a día. Los sistemas van alejándose de sus especificaciones funcionalidad a funcionalidad, mientras se añade un parche tras otro a una porción del código hasta que ya no queda nada del original. A menudo, esa acumulación de cosas pequeñas es la que acaba con la moral y con los equipos.

Truco 7. Recuerde el panorama general.

Nunca hemos probado esto, de verdad, pero dicen que, si coges una rana y la sueltas en agua hirviendo, saltará fuera del recipiente. Sin embargo, si la dejas en una cazuela con agua fría y después vas calentándola de manera gradual, la rana no notará el aumento lento de la temperatura y se quedará quieta hasta que esté cocinada.

Observe que el problema de la rana es diferente al de las ventanas rotas que hemos visto en el tema 3, “Entropía del software”. En la teoría de las ventanas rotas, la gente pierde la voluntad de luchar contra la entropía porque percibe que a nadie más le importa. La rana no nota el cambio, sin más.

No sea como la rana imaginaria. No pierda de vista el panorama general. Revise constantemente lo que pasa a su alrededor, no solo lo que está haciendo usted a nivel individual.

Las secciones relacionadas incluyen

- Tema 1, “Es su vida”.
- Tema 38, “Programar por casualidad”.

Retos

- Mientras revisaba un borrador de la primera edición, John Lakos planteó la siguiente cuestión: los soldados engañaron a los aldeanos de manera progresiva, pero el cambio que catalizaron fue bueno para todos. Sin embargo, al engañar a la rana de forma progresiva, le hacemos daño. ¿Puede determinar si está haciendo sopa de piedras o sopa de ranas cuando intenta catalizar un cambio? ¿Es la decisión subjetiva u objetiva?
- Rápido, sin mirar, ¿cuántas luces hay encima de usted? ¿Y cuántas hay en la sala? ¿Cuánta gente? ¿Hay algo fuera de contexto, algo que parezca que no debería estar ahí? Este es un ejercicio de consciencia situacional, una técnica practicada a menudo por personas que van desde *Boy* y *Girl Scouts* a los SEAL de la Armada de EE. UU. Acostúmbrese a prestar atención y a ser consciente de lo que pasa a su alrededor. Después, haga lo mismo en su proyecto.

5 Software lo bastante bueno

Lo bueno se malogra queriendo mejorarlo.

—Shakespeare, El rey Lear 1.4.

Hay un chiste (un poco) viejo sobre una empresa que hace un pedido de 100.000 CI a un fabricante japonés. Parte de la especificación era la tasa de defectos: un chip de cada 10.000. Unas semanas después, llega el pedido: una caja grande que contiene miles de CI y una caja pequeña que contiene

solo diez. Pegada en la caja pequeña hay una etiqueta que dice: “Estos son los diez defectuosos”.

Ojalá tuviésemos de verdad este tipo de control sobre la calidad. Pero el mundo real no nos deja producir mucho que sea realmente perfecto, sobre todo software libre de errores. El tiempo, la tecnología y el temperamento conspiran contra nosotros.

Sin embargo, esto no tiene por qué ser frustrante. Como describió Ed Yourdon en un artículo en *IEEE Software*, *When good-enough software is best* [You95], puede disciplinarse para escribir software que sea lo bastante bueno, para sus usuarios, para las personas que tendrán que mantenerlo en el futuro y para su propia tranquilidad. Descubrirá que es más productivo y sus usuarios están más contentos. Y puede que descubra que sus programas son en realidad mejores por su incubación más corta.

Antes de seguir, tenemos que matizar lo que vamos a decir. La frase “lo bastante bueno” no implica código chapucero o mal producido. Todos los sistemas deben cumplir los requisitos de sus usuarios para tener éxito, y deben ajustarse a unos estándares básicos de rendimiento, privacidad y seguridad. Lo que defendemos es que se dé a los usuarios la oportunidad de participar en el proceso de decidir si lo que hemos producido es lo bastante bueno para sus necesidades.

Implique a sus usuarios en la compensación

Por lo general, escribirá software para otras personas. A menudo, se acordará de averiguar lo que quieren.[6] Pero ¿les pregunta alguna vez lo bueno que quieren que sea su software? A veces, no habrá opción. Si está trabajando en marcapasos, sistemas de piloto automático o una biblioteca de bajo nivel que va a divulgarse ampliamente, los requisitos serán más rigurosos y sus opciones serán más limitadas.

Sin embargo, si está trabajando en un producto totalmente nuevo, tendrá restricciones diferentes. Los responsables del marketing tendrán promesas que cumplir, los usuarios finales pueden haber hecho planes en función de una fecha de entrega y, desde luego, su empresa tendrá restricciones respecto al flujo de caja. Sería poco profesional ignorar los requisitos de estos usuarios solo para añadir funcionalidades nuevas al programa o para

pulir el código una vez más. No estamos abogando por el pánico: sería igual de poco profesional prometer plazos imposibles y recortar elementos de ingeniería básicos para cumplir esos plazos.

El alcance y la calidad del sistema que producimos deberían describirse como parte de los requisitos de ese sistema.

Truco 8. Convierta la calidad en una cuestión de los requisitos.

A menudo, se verá en situaciones en las que se requiere una compensación. Sorprendentemente, muchos usuarios preferirían utilizar software sin pulir del todo hoy que esperar un año para tener la versión brillante y con toda la parafernalia (y, en realidad, puede que lo que necesiten de aquí a un año sea totalmente diferente, de todas formas). Muchos departamentos de TI con presupuestos muy ajustados estarán de acuerdo. A menudo, un software genial hoy es preferible a la fantasía de un software perfecto mañana. Si da a sus usuarios algo con lo que jugar pronto, con frecuencia su *feedback* le llevará a conseguir una solución final mejor (véase el tema 12, “Balas trazadoras”).

Saber cuándo parar

En algunos aspectos, programar es como pintar. Empezamos con un lienzo en blanco y determinadas materias primas básicas. Utilizamos una combinación de ciencia, arte y oficio para determinar qué hacer con ellas. Esbozamos una forma general, pintamos el entorno subyacente y, después, añadimos los detalles. Damos un paso atrás todo el tiempo con ojo crítico para ver lo que hemos hecho. De vez en cuando, desechamos un lienzo y empezamos otra vez.

Pero los artistas dirán que todo el trabajo duro se verá arruinado si no sabemos cuándo parar. Si añadimos capa tras capa, detalle tras detalle, el cuadro se pierde en la pintura.

No arruine un buen programa tratando de embellecerlo o refinarlo en exceso. Siga adelante y deje que su código se mantenga por sí mismo durante un tiempo. Puede que no sea perfecto. No se preocupe: nunca

podría ser perfecto. (En el capítulo 7, “Mientras escribe código”, veremos filosofías para desarrollar código en un mundo imperfecto).

Las secciones relacionadas incluyen

- Tema 45, “El pozo de los requisitos”.
- Tema 46, “Resolver rompecabezas imposibles”.

Retos

- Fíjese en las herramientas de software y los sistemas operativos que utiliza con regularidad. ¿Puede encontrar cualquier prueba de que estas organizaciones o desarrolladores se sienten cómodos enviando software que saben que no es perfecto? Como usuario, ¿preferiría (1) esperar a que eliminen todos los errores, (2) tener un software complejo y aceptar algunos errores o (3) optar por un software más simple con menos errores?
- Considere el efecto de la modularización en la entrega de software. ¿Llevaría más o menos tiempo conseguir un bloque de software monolítico con acoplamiento fuerte para la calidad requerida en comparación con un sistema diseñado como módulos de microservicios con acoplamiento muy débil? ¿Cuáles son las ventajas y las desventajas de cada enfoque?
- ¿Se le ocurre algún software popular que tenga un exceso de funcionalidades innecesarias? Es decir, un software que contenga muchas más funcionalidades de las que se usarán jamás, cada una de ellas generando más posibilidades de que haya errores y puntos débiles en la seguridad, y haciendo que las funcionalidades que sí se usan sean más difíciles de encontrar y gestionar. ¿Corre usted mismo el peligro de caer en esta trampa?

6 Su cartera de conocimientos

Invertir en conocimientos produce siempre los mejores beneficios.

—Benjamin Franklin.

Ah, el viejo Ben Franklin, siempre con una homilía concisa a mano. Bueno, si pudiésemos irnos pronto a la cama y madrugar, seríamos magníficos programadores, ¿no? Según un proverbio inglés, el pájaro madrugador se lleva el gusano, pero ¿qué pasa con el gusano madrugador?

Sin embargo, en este caso, Ben da en el clavo. Nuestro conocimiento y nuestra experiencia son nuestros activos profesionales más importantes en el día a día.

Por desgracia, son activos con fecha de caducidad.[7] Nuestros conocimientos se quedan obsoletos a medida que se desarrollan técnicas, lenguajes y entornos nuevos. Las fuerzas del mercado cambiante pueden hacer que nuestra experiencia quede desfasada o sea irrelevante. Teniendo en cuenta el ritmo cada vez más veloz del cambio en nuestra sociedad tecnológica, esto puede ocurrir con bastante rapidez.

A medida que disminuye el valor de nuestro conocimiento, también se reduce nuestro valor para nuestra empresa o nuestro cliente. Queremos evitar que eso suceda.

Nuestra capacidad para aprender cosas nuevas es nuestro activo estratégico más importante. Pero ¿cómo aprendemos a aprender, y cómo sabemos qué aprender?

Su cartera de conocimientos

Nos gusta pensar que todos los hechos que los programadores saben acerca de la informática, los dominios de aplicación en los que trabajan y toda su experiencia son como sus carteras de conocimientos. Gestionar una cartera de conocimientos es muy similar a gestionar una cartera de valores:

1. Los inversores serios invierten con regularidad, como costumbre.
2. La diversificación es la clave del éxito a largo plazo.
3. Los inversores inteligentes mantienen un equilibrio en sus carteras entre inversiones conservadoras e inversiones de alto riesgo con recompensas elevadas.
4. Los inversores intentan comprar por precios bajos y vender por precios altos para obtener la máxima ganancia.

5. Las carteras deberían revisarse y volver a equilibrarse de forma periódica.

Para tener éxito en su carrera, debe invertir en su cartera de conocimientos usando estas mismas directrices.

La buena noticia es que gestionar este tipo de inversión es una habilidad como cualquier otra, así que puede aprenderse. El truco es obligarse a hacerlo al principio y convertirlo en un hábito. Desarrolle una rutina y sígala hasta que su cerebro la interiorice. En ese punto, verá que absorbe conocimientos nuevos de manera automática.

La creación de su cartera

- Invierta con regularidad
  Al igual que ocurre con las inversiones financieras, debe invertir en su cartera de conocimientos con regularidad, aunque sea solo una cantidad pequeña. El hábito es tan importante como la suma, así que planee utilizar un momento y un lugar de forma consistente, lejos de interrupciones. En la siguiente sección veremos algunos objetivos de ejemplo.
- Diversifique
  Cuantas más cosas diferentes sepa, más valioso será. Como punto de partida, necesita conocer los pormenores de la tecnología concreta con la que esté trabajando en ese momento, pero no se quede ahí. El mundo de la informática cambia con rapidez; lo que hoy se considera tecnología punta puede ser casi inútil (o, al menos, tener poca demanda) mañana. Cuantas más tecnologías domine con comodidad, mejor podrá adaptarse al cambio. Y no olvide todas las demás habilidades que necesita, incluyendo aquellas que no pertenecen a áreas técnicas.
- Gestione el riesgo
  La tecnología existe a lo largo de un espectro que va desde estándares arriesgados con recompensas potencialmente altas a estándares con poco riesgo y recompensas bajas. No es aconsejable invertir todo su dinero en acciones de alto riesgo que pueden hundirse de repente, ni tampoco invertirlo todo de forma conservadora y dejar pasar posibles

oportunidades. No ponga todos los huevos técnicos en la misma cesta.

- Compre barato, venda caro
  Aprender una tecnología emergente antes de que se popularice puede ser tan difícil como encontrar acciones infravaloradas, pero el beneficio puede ser igual de provechoso. Aprender Java cuando se introdujo por primera vez y era desconocido pudo haber supuesto un riesgo en aquel momento, pero compensó con creces a los primeros en adoptarlo cuando se convirtió en un pilar de la industria más tarde.
- Revise y vuelva a equilibrar
  Esta es una industria muy dinámica. Esa tecnología punta que empezó a investigar el mes pasado puede estar muerta a estas alturas. Quizá necesita repasar esa tecnología de base de datos que no ha utilizado en un tiempo. O tal vez tendría más opciones para ese nuevo puesto de trabajo si probase ese otro lenguaje...

De todas estas directrices, la más importante es la más fácil de hacer.

Truco 9. Invierta con regularidad en su cartera de conocimientos.

Objetivos

Ahora que tenemos algunas directrices acerca de qué y cuándo sumar a nuestra cartera de conocimientos, ¿cuál es la mejor forma de adquirir el capital intelectual con el que financiar nuestra cartera? Veamos algunas sugerencias:

- Aprenda al menos un lenguaje nuevo cada año
  Diferentes lenguajes resuelven el mismo problema de formas distintas. Al aprender varios enfoques diferentes, puede ayudar a expandir sus ideas y evitar quedarse atrapado en la rutina. Además, aprender muchos lenguajes es fácil gracias a la gran cantidad de software disponible de manera gratuita.
- Lea un libro técnico al mes
  Aunque hay una saturación de ensayos cortos y, de vez en cuando, respuestas fiables en Internet, para obtener un entendimiento

profundo necesita libros largos. Explore las librerías en busca de libros técnicos sobre temas interesantes relacionados con su proyecto actual.[8] Una vez que adquiera la costumbre, lea un libro al mes. Cuando domine las tecnologías que está utilizando en este momento, expanda sus horizontes y estudie algo que no tenga relación con su proyecto.

- Lea también libros que no sean técnicos
  Es importante recordar que los ordenadores los utilizan las personas, personas cuyas necesidades está intentando satisfacer. Trabaja con personas, le contratan personas y le hackean personas. No olvide el elemento humano de la ecuación, porque eso requiere un conjunto de habilidades totalmente diferente (es irónico que las llamemos "habilidades blandas", porque en realidad son bastante difíciles de dominar).
- Tome clases
  Busque cursos interesantes en una institución o universidad local o en línea, o quizá en la siguiente feria o conferencia a la que asista.
- Participe en grupos de usuarios y encuentros locales
  El asilamiento puede ser mortal para su carrera; averigüe en qué está trabajando la gente fuera de su empresa. No vaya solo a escuchar: participe de forma activa.
- Experimente con entornos diferentes
  Si solo ha trabajado en Windows, pase algo de tiempo con Linux. Si solo ha utilizado `makefiles` y un editor, pruebe un IDE sofisticado con funcionalidades innovadoras y viceversa.
- Esté al día
  Lea noticias y publicaciones en línea sobre tecnología diferente a la de su proyecto actual. Es una manera genial de descubrir qué experiencias están teniendo con ella otras personas, la jerga particular que utilizan, etc.

Es importante seguir investigando. Una vez que se sienta cómodo con un nuevo lenguaje o una tecnología, siga avanzando. Aprenda otros.

No importa si nunca usa ninguna de esas tecnologías en un proyecto, ni siquiera si las pone en su currículo. El proceso de aprendizaje expandirá su mente, abriéndole a nuevas posibilidades y nuevas maneras de hacer las

cosas. La polinización cruzada de ideas es importante; intente aplicar las lecciones que ha aprendido a su proyecto actual. Incluso si su proyecto no utiliza esa tecnología, quizá pueda tomar prestadas algunas ideas. Familiarícese con la orientación a objetos, por ejemplo, y escribirá programas por procedimientos de forma distinta. Entienda el paradigma de la programación funcional y escribirá el código orientado a objetos de manera diferente, etc.

Oportunidades para el aprendizaje

Como está leyendo con tanta voracidad, está al tanto de los últimos desarrollos que se producen en su campo (cosa que no es fácil) y alguien le hace una pregunta. Usted no tiene ni idea de cuál es la respuesta, y lo admite sin problema.

No se quede ahí. Tómese como un desafío personal encontrar la respuesta. Pregunte a otros. Busque en Internet, incluyendo las partes académicas, no solo las de los consumidores.

Si no puede encontrar la respuesta usted mismo, averigüe quién puede. No lo deje correr. Hablar con otras personas le ayudará a crear su red personal y puede que se sorprenda a sí mismo encontrando soluciones para otros problemas no relacionados por el camino. Y esa vieja cartera sigue creciendo y creciendo...

Toda esta lectura y esta investigación llevan tiempo, y ya hay poco tiempo de por sí, así que necesita planificación. Tenga siempre algo a mano para leer en los ratos muertos. El tiempo que se pasa en la sala de espera del médico o el dentista puede ser una gran oportunidad para ponerse al día, pero asegúrese de llevarse su propio lector de libros electrónicos o puede que acabe hojeando un artículo sobado de 1973 sobre Papúa Nueva Guinea.

Pensamiento crítico

La última cuestión importante es pensar de manera crítica en lo que lee y escucha. Tiene que asegurarse de que los conocimientos que guarda en su cartera son exactos y no están influidos por el bombo que les dan los distribuidores o los medios. Tenga cuidado con los fanáticos que insisten en

que su dogma proporciona la única respuesta; puede ser o no ser aplicable a usted y a su proyecto. Nunca subestime el poder del comercio. Que un motor de búsqueda en la web muestre primero un resultado no significa que sea el más adecuado; el proveedor de contenido puede pagar para aparecer en lo más alto. Que una librería coloque un libro en un lugar prominente no quiere decir que sea un buen libro, ni siquiera popular; pueden haber pagado para que se ponga ahí.

Truco 10. Analice de forma crítica lo que lea y escuche.

El pensamiento crítico es una disciplina entera en sí mismo, y le animamos a leer y estudiar todo lo que pueda al respecto. Mientras tanto, aquí tiene algo para empezar con algunas preguntas para que las responda y piense en ellas.

- Pregunte los “cinco porqués”
  Uno de los trucos favoritos del asesoramiento: pregunte “¿por qué?” al menos cinco veces. Haga una pregunta y obtenga una respuesta. Profundice preguntando “¿por qué?”. Repita como si fuese un niño quisquilloso de cuatro años (pero uno educado). Puede que consiga acercarse más a la raíz del asunto de ese modo.
- ¿A quién beneficia esto?
  Puede que suene cínico, pero seguir el dinero puede ser una ruta muy útil para analizar. Los beneficios para otra persona u otra organización pueden estar o no estar en consonancia con los suyos.
- ¿Cuál es el contexto?
  Todo ocurre en su propio contexto, y esa es la razón por la que las soluciones “de talla única” no suelen valer para todos. Piense en un artículo o libro que venda “la mejor práctica”. Algunas preguntas buenas a tener en cuenta son: ¿mejor para quién?, ¿cuáles son prerrequisitos?, ¿cuáles son las consecuencias a corto y largo plazo?
- ¿Cuándo o dónde funcionaría esto?
  ¿En qué circunstancias? ¿Es demasiado tarde? ¿Demasiado pronto? No se detenga en el pensamiento de primer orden (lo que pasará a

continuación), utilice pensamiento de segundo orden: ¿qué pasará después de eso?

- ¿Por qué es esto un problema?
  ¿Hay un modelo subyacente? ¿Cómo funciona el modelo subyacente?

Por desgracia, ya hay muy pocas respuestas simples. Pero con su amplia cartera, y aplicando cierto análisis crítico al torrente de artículos técnicos que va a leer, podrá entender las respuestas complejas.

Las secciones relacionadas incluyen

- Tema 1, “Es su vida”.
- Tema 22, “Cuadernos de bitácora de ingeniería”.

Retos

- Empiece a aprender un lenguaje nuevo esta semana. ¿Ha programado siempre en el mismo viejo lenguaje? Pruebe Clojure, Elixir, Elm, F#, Go, Haskell, Python, R, ReasonML, Ruby, Rust, Scala, Swift, TypeScript, o cualquier otro que le llame la atención o que parezca que le va a gustar.[9]
- Empiece a leer un libro nuevo (¡pero termine este primero!). Si está realizando una implementación y una creación de código muy profundas, lea un libro sobre diseño y arquitectura. Si está llevando a cabo un diseño de alto nivel, lea un libro sobre técnicas de creación de código.
- Salga y hable de tecnología con personas que no estén involucradas en su proyecto actual o que no trabajen para la misma empresa. Desarrolle una red de contactos en la cafetería de la empresa o busque a otros entusiastas en un encuentro local.

7 ¡Comuníquese!

Es mejor ser examinado que ignorado.

—Mae West, No es pecado, 1934.

Quizá podamos aprender una lección de la señorita West. No es solo lo que tenemos, sino también como lo presentamos. Tener las mejores ideas, el código más perfecto o el pensamiento más pragmático al final no sirve de nada a menos que podamos comunicarnos con otras personas. Una idea buena está huérfana sin una comunicación efectiva.

Como desarrolladores, tenemos que comunicarnos a muchos niveles. Pasamos horas en reuniones, escuchando y hablando. Trabajamos con usuarios finales, intentando entender sus necesidades. Escribimos código, que comunica nuestras intenciones a una máquina y documenta nuestro pensamiento para futuras generaciones de desarrolladores. Escribimos propuestas y circulares solicitando y justificando recursos, informando de nuestro estado y sugiriendo nuevos enfoques. Y trabajamos a diario con nuestros equipos para defender nuestras ideas, modificar prácticas existentes y sugerir otras nuevas. Pasamos gran parte del día comunicándonos, así que tenemos que hacerlo bien.

Trate su lengua materna como cualquier otro lenguaje de programación. Escriba lenguaje natural como escribiría código: honre el principio DRY (*don't repeat yourself*, no te repitas), ETC (*easy to change*, fácil de cambiar,) la automatización, etc. (Veremos los principios de diseño DRY y ETC en el siguiente capítulo).

Truco 11. Su lengua materna es solo otro lenguaje de programación.

Hemos preparado una lista de ideas adicionales que nos parecen útiles.

Conozca a su público

Solo está comunicándose si está transmitiendo lo que quiere transmitir, hablar sin más no basta. Para ello, tiene que entender las necesidades, los intereses y las capacidades de su público. Todos hemos estado en reuniones en las que un friki del desarrollo hace que al vicepresidente de marketing se le cierren los ojos con un largo monólogo acerca de los méritos de alguna tecnología arcana. Eso no es comunicar: es solo hablar, y es una pesadez.

Digamos que quiere cambiar su sistema de monitorización remoto para usar un bróker de mensajería de terceros para difundir notificaciones de

estado. Puede presentar esta actualización de muchas maneras distintas, dependiendo de su público. Los usuarios finales apreciarán que ahora sus sistemas puedan interactuar con otros servicios que usen el bróker. El departamento de marketing podrá utilizar ese hecho para impulsar las ventas. Los directores de desarrollo y operaciones estarán contentos porque el cuidado y el mantenimiento de esa parte del sistema ahora es problema de otros. Por último, los desarrolladores pueden disfrutar adquiriendo experiencia con nuevas API, y puede que incluso encuentren nuevos usos para el bróker de mensajería. Al hacer el discurso apropiado para cada grupo, conseguirá que todos se emocionen con el proyecto.

Como ocurre con todas las formas de comunicación, aquí el truco está en reunir *feedback*. No se siente sin más a esperar preguntas: pídalas. Fíjese en el lenguaje corporal y en las expresiones faciales. Uno de los supuestos de la programación neurolingüística es: "El significado de la comunicación es la respuesta que recibes". Mejore de forma continua el conocimiento de su público a medida que se comunica.

Sepa lo que quiere decir

Es probable que la parte más difícil de los estilos más formales de comunicación utilizados en los negocios sea determinar con exactitud lo que se quiere decir. A menudo, los escritores de ficción planean el argumento de sus libros con detalle antes de empezar, pero a las personas que escriben documentos técnicos suele gustarles sentarse delante del teclado, escribir:

1. Introducción

y empezar a teclear lo que les va pasando por la cabeza.

Planifique lo que quiere decir. Escriba un esquema y, después, pregúntese: "¿Comunica esto lo que quiero expresar a mi público de una manera que funcione para él?". Perfecciónelo hasta que lo haga.

Este enfoque no funciona solo para documentos. Cuando se enfrente a una reunión importante o a una charla con uno de los clientes principales, anote las ideas que desea comunicar y planee un par de estrategias para asegurarse de que se entienden.

Ahora que sabe lo que quiere su público, entréguelo.

Elija el momento

Son las seis de la tarde del viernes de una semana en la que los auditores han visitado la oficina. El hijo más pequeño de su jefa está en el hospital, está lloviendo a mares y está claro que el viaje de vuelta a casa va a ser una pesadilla. Es probable que no sea un buen momento para pedirle una actualización de la memoria de su portátil.

Como parte de la comprensión de lo que necesita oír su público, tiene que averiguar cuáles son sus prioridades. Aborde a una directora a la que su jefe acaba de regañar por un código fuente perdido y tendrá a una oyente más receptiva a sus ideas sobre repositorios de código fuente. Haga que lo que dice sea relevante respecto al momento, además de respecto al contenido. A veces, lo único que hace falta es una simple pregunta: “¿Es un buen momento para hablar de...?”.

Elija un estilo

Ajuste el estilo de su discurso para que se adapte a su público. Algunas personas quieren una sesión informativa de tipo “solo los hechos”. Otros prefieren una charla larga y variada antes de entrar en materia. ¿Cuál es su nivel de habilidad y experiencia en esta área? ¿Son expertos? ¿Novatos? ¿Hay que explicárselo todo paso a paso o prefieren en resumen? En caso de duda, pregúnteles.

Recuerde, sin embargo, que usted es la mitad de la transacción comunicativa. Si alguien le dice que necesita un párrafo que describa algo y usted sabe que no puede hacerse en menos de varias páginas, dígaselo. No olvide que ese tipo de *feedback* también es una forma de comunicación.

Dele un buen aspecto

Sus ideas son importantes. Se merecen un vehículo con un buen aspecto que las transmita al público.

Demasiados desarrolladores (y sus directores) se concentran solo en el contenido cuando producen documentos escritos. A nosotros nos parece un error. Cualquier chef (o espectador del canal de cocina) le dirá que puede darlo todo en la cocina durante horas solo para acabar echando por tierra sus esfuerzos con una presentación pobre.

Hoy en día, no hay excusa para producir documentos impresos con un aspecto pobre. El software moderno puede producir unos resultados impresionantes, al margen de si estamos utilizando Markdown o un procesador de texto. Solo hace falta aprender algunos comandos básicos. Si usa un procesador de texto, utilice sus hojas de estilo por coherencia. (Puede que su empresa ya tenga hojas de estilo definidas que pueda utilizar). Aprenda a configurar encabezados y pies de página. Fíjese en los documentos de muestra incluidos en el paquete para sacar ideas sobre el estilo y el diseño. Revise la ortografía, primero de forma automática y, después, a mano. “Al fan y al cava, ay errores de ortografía que el correcto no pueden detectar”.

Involucre a su público

A menudo, nos encontramos con que los documentos que producimos acaban siendo menos importantes que el proceso por el que pasamos para producirlo. Si es posible, involucre a sus lectores con los primeros borradores de su documento. Recoja su *feedback* y hágales preguntas. Creará una buena relación de trabajo y es probable que produzca un documento mejor en el proceso.

Sepa escuchar

Hay una técnica que debe utilizar si quiere que las personas le escuchen: escúchelas a ellas. Incluso si esta es una situación en la que usted tiene toda la información, incluso si es una reunión formal en la que esté de pie ante 20 jefes, si no les escucha, no le escucharán a usted.

Anime a la gente a hablar haciendo preguntas o pídales que vuelvan a exponer lo que se ha dicho con sus propias palabras. Convierta la reunión

en un diálogo y transmitirá su propósito de manera más efectiva. Quién sabe, a lo mejor incluso aprende algo.

Responda a la gente

Si hace una pregunta a alguien, le parece una falta de educación que no le respondan. Pero ¿con cuánta frecuencia no responde usted a alguien cuando le envían un correo o una circular solicitando información o pidiéndole que haga algo? En el ajetreo de la vida diaria, es fácil olvidarse de hacerlo. Responda siempre a los correos y mensajes de voz, incluso aunque la respuesta sea solo: “Te respondo luego”. Mantener a otros informados hace que sean mucho más comprensivos con la metedura de pata ocasional y hace que sientan que no se ha olvidado de ellos.

Truco 12. Importa lo que dice y también cómo lo dice.

A menos que trabaje en un vacío, debe ser capaz de comunicarse. Cuanto más efectiva sea esa comunicación, más influyente se volverá usted.

Documentación

Por último, está la cuestión de la comunicación a través de la documentación. Por lo general, los desarrolladores no piensan mucho en la documentación. En el mejor de los casos, es una desafortunada necesidad; en el peor, se trata como una tarea de prioridad baja con la esperanza de que los directores se olvidarán de ella al final del proyecto.

Los desarrolladores pragmáticos abrazan la documentación como parte integral del proceso de desarrollo global. La escritura de la documentación puede facilitarse si no se duplican los esfuerzos ni se pierde el tiempo y la documentación se mantiene a mano, en el propio código. De hecho, nos conviene aplicar todos nuestros principios pragmáticos a la documentación, además de al código.

Truco 13. Integre la documentación, no la acople como algo extra.

Es fácil producir documentación con buen aspecto a partir de los comentarios en el código fuente y recomendamos añadir comentarios a módulos y funciones exportadas para dar a otros desarrolladores algo de ayuda cuando vayan a utilizarlo.

Sin embargo, eso no significa que estemos de acuerdo con quienes dicen que cada función, estructura de datos, declaración de tipo, etc., necesita su propio comentario. Este tipo de comentario mecánico en realidad hace que sea más difícil mantener el código: ahora hay dos cosas que actualizar cuando hacemos un cambio. Así pues, limite sus comentarios que no sean de API a describir por qué se hace algo, su propósito y objetivo. El código ya muestra cómo se hace, así que comentarlo es redundante y una violación del principio DRY.

Comentar código fuente le da la oportunidad perfecta de documentar todos esos fragmentos esquivos de un proyecto que no pueden documentarse en ninguna otra parte: compensaciones de ingeniería, por qué se han tomado esas decisiones, qué otras alternativas había, etc.

Resumen

- Sepa lo que quiere decir.
- Conozca a su público.
- Elija el momento.
- Elija un estilo.
- Dele un buen aspecto.
- Involucre a su público.
- Sepa escuchar.
- Responda a la gente.
- Mantenga el código y la documentación juntos.

Las secciones relacionadas incluyen

- Tema 15, “Estimaciones”.
- Tema 18, “Edición potente”.
- Tema 45, “El pozo de los requisitos”.
- Tema 49, “Equipos pragmáticos”.

Retos

- Hay muchos libros estupendos que contienen secciones sobre la comunicación dentro de los equipos, incluyendo *The Mythical Man-Month: Essays on Software Engineering* [Bro96] y *Peopleware: Productive Projects and Teams* [DL13]. Esfuércese por intentar leer esos dos en los próximos 18 meses. Además, *Cerebros de dinosaurio. Cómo tratar con personas imposibles en el trabajo* [BR89] habla del bagaje emocional que todos llevamos al entorno laboral.
- La próxima vez que tenga que hacer una presentación o escribir un memorándum defendiendo alguna postura, pruebe a trabajar siguiendo los consejos de la sección anterior antes de empezar. Identifique de manera explícita a su público y lo que necesita comunicar. Si es apropiado, hable con el público después y compruebe lo exacta que ha sido la valoración que había hecho de sus necesidades.

Comunicación en línea

Todo lo que hemos dicho de la comunicación por escrito se aplica del mismo modo al correo electrónico, publicaciones en redes sociales, blogs, etc. El correo electrónico en particular ha evolucionado hasta el punto en que es uno de los pilares de las comunicaciones corporativas; se utiliza para discutir contratos, resolver disputas y como prueba en un juicio. Pero, por alguna razón, personas que jamás enviarían un documento en papel manoseado no tienen reparos en mandar correos electrónicos incoherentes y con un aspecto horrible por todo el mundo.
Nuestros consejos son sencillos:

- Revise el texto antes de pulsar ENVIAR.
- Compruebe la ortografía y busque percances accidentales del corrector automático.
- Mantenga el formato simple y claro.
- Mantenga las citas al mínimo. A nadie le gusta volver a recibir su propio correo de 100 líneas con un “Estoy de acuerdo” añadido al final.
- Si cita el correo de otra persona, asegúrese de mencionar quién es y cítelo en el propio texto (en vez de como un adjunto). Lo mismo se aplica cuando se cita en plataformas de redes sociales.
- No lance ataques ni se comporte como un trol a menos que quiera que ese comportamiento acabe volviendo y causándole problemas más tarde. Si es algo que no le diría a una persona cara a cara, no lo diga en línea.

- Compruebe la lista de destinatarios antes de enviar. Se ha convertido en un cliché criticar al jefe en el correo del departamento sin darse cuenta de que él está en la lista de correo en copia. Mejor aún, no critique al jefe por correo electrónico.

Como han descubierto innumerables empresas grandes y políticos, los correos electrónicos y las publicaciones en redes sociales duran para siempre. Intente poner la misma atención y cuidado en los correos que en cualquier memorándum o informe que escriba.

---

[1] http://wiki.c2.com/?ChangeYourOrganization.

[2] Puede ver, por ejemplo, un buen metaanálisis en *Trust and team performance: A meta-analysis of main effects, moderators, and covariates*, http://dx.doi.org/10.1037/apl0000110.

[3] Consulte *The police and neighborhood safety* [WH82].

[4] Consulte *Contagious depression: Existence, specificity to depressed symptoms, and the role of reassurance seeking* [Joi94].

[5] Mientras hace esto, puede que le resulte reconfortante la cita atribuida a la contralmirante Dra. Grace Hopper: "Es más fácil pedir perdón que conseguir permiso".

[6] ¡Se supone que eso era un chiste!

[7] Un activo con fecha de caducidad es algo cuyo valor disminuye con el tiempo. Entre los ejemplos se incluyen un almacén lleno de plátanos o una entrada para un partido de béisbol.

[8] Puede que nosotros no seamos imparciales, pero hay una buena selección en https://pragprog.com.

[9] ¿Nunca ha oído hablar de estos lenguajes? Recuerde: el conocimiento es un activo con fecha de caducidad, y lo mismo ocurre con la tecnología popular. La lista de lenguajes nuevos y experimentales punteros es muy diferente de la que había en la primera edición y es probable que sea diferente otra vez para cuando lea esto. Razón de más para seguir aprendiendo.

2
Un enfoque pragmático

Existen algunos consejos y trucos que se aplican a todos los niveles del desarrollo de software, procesos que son prácticamente universales e ideas que son casi axiomáticas. Sin embargo, estos enfoques rara vez se documentan como tales; lo más habitual es encontrarlos escritos como frases sueltas en discusiones sobre diseño, gestión de proyectos o creación de código. Pero, para su conveniencia, hemos reunido aquí estas ideas y procesos.

El primer tema, y quizá el más importante, llega al corazón del desarrollo de software: “La esencia del buen diseño”. Todo lo demás sigue a partir de ahí.

Las dos secciones siguientes, “DRY: los males de la duplicación” y “Ortogonalidad”, están bastante relacionados. El primero le advierte que no duplique la información a lo largo del sistema y el segundo que no divida una sola porción de información entre múltiples componentes del sistema.

A medida que aumenta el ritmo del cambio, se va haciendo cada vez más difícil conseguir que nuestras aplicaciones sigan siendo relevantes. En “Reversibilidad”, echaremos un vistazo a algunas técnicas que ayudan a aislar nuestros proyectos de su entorno cambiante.

Las siguientes dos secciones también están relacionadas. En “Balas trazadoras”, hablamos de un estilo de desarrollo que permite reunir requisitos, probar diseños e implementar código al mismo tiempo. Es la única forma de seguir el ritmo de la vida moderna.

“Prototipos y notas en *post-its*” muestra cómo utilizar prototipos para probar arquitecturas, algoritmos, interfaces e ideas. En el mundo moderno, es crucial probar ideas y recibir *feedback* antes de comprometerse con ellas por completo.

Mientras la informática madura con lentitud, los diseñadores están produciendo lenguajes de nivel cada vez más alto. Aunque todavía no se ha inventado el compilador que acepte “hazlo así”, en “Lenguajes de dominio”

presentamos algunas sugerencias más modestas que puede implementar usted mismo.

Por último, todos trabajamos en un mundo con tiempo y recursos limitados. Puede sobrevivir mejor a esta escasez (y tener a sus jefes o clientes contentos) si mejora su capacidad para determinar cuánto tiempo le llevará hacer las cosas, que es de lo que hablaremos en “Estimaciones”.

Tenga en mente estos principios fundamentales durante el desarrollo y escribirá código que será mejor, más rápido y más fuerte. Incluso puede hacer que parezca fácil.

8 La esencia del buen diseño

El mundo está lleno de gurús y expertos más que dispuestos a transmitir su sabiduría adquirida tras un gran esfuerzo acerca de Cómo Diseñar Software. Hay acrónimos, listas (que parecen preferir cinco entradas), patrones, diagramas, vídeos, charlas y (por ser Internet como es), probablemente una serie guay sobre la ley de Demeter explicada mediante danza interpretativa.

Y nosotros, los amables autores, también somos culpables de esto. Pero nos gustaría reparar el daño causado explicando algo que solo nos ha empezado a parecer evidente hace poco. Primero, la afirmación general:

Truco 14. Un diseño bueno es más fácil de cambiar que un diseño malo.

Una cosa está bien diseñada si se adapta a las personas que la utilizan. Para el código, eso significa que debe adaptarse cambiando, así que creemos en el principio ETC: *Easier to Change*, más fácil de cambiar. ETC. Eso es todo. Por lo que sabemos, cualquier principio de diseño que existe es un caso especial del ETC.

¿Por qué es bueno el desacoplamiento? Porque al aislar los intereses hacemos que sea más fácil cambiar. ETC.

¿Por qué es útil el principio de responsabilidad única? Porque un cambio en los requisitos se corresponde con un cambio en un solo módulo. ETC.

¿Por qué es importante el momento de poner nombres? Porque los buenos nombres hacen que el código sea más fácil de leer, y hay que leerlo para cambiarlo. ¡ETC!

ETC es un valor, no una regla

Los valores son cosas que nos ayudan a tomar decisiones: ¿debería hacer esto o aquello? A la hora de pensar en el software, ETC es una guía que nos ayuda a elegir entre varios caminos. Al igual que todos los demás valores, debería estar flotando en su pensamiento consciente, empujándole de manera sutil en la dirección correcta.

Pero ¿cómo hacemos que ocurra eso? Nuestra experiencia es que requiere cierto refuerzo consciente inicial. Puede que se pase una semana, más o menos, preguntándose de manera deliberada si lo que acaba de hacer ha hecho que el sistema general sea más fácil o más difícil de cambiar. Hágalo cuando guarde un archivo. Hágalo cuando escriba una prueba. Hágalo cuando arregle un fallo.

Hay una premisa implícita en ETC. Asume que una persona puede decir cuál entre muchos caminos será más fácil de cambiar en el futuro. En muchas ocasiones, el sentido común estará en lo cierto, y puede hacer conjeturas fundamentadas.

A veces, sin embargo, no tendrá ni idea. No pasa nada. En esos casos, creemos que puede hacer dos cosas.

Primero, dado que no está seguro de qué forma adoptará el cambio, siempre puede ir por el camino definitivo del “fácil de cambiar”: intente hacer que lo que escriba sea sustituible. De ese modo, pase lo que pase en el futuro, esta porción de código no será un obstáculo en el camino. Parece algo extremo, pero, en realidad, es lo que debería estar haciendo todo el tiempo, en cualquier caso. Se trata solo de pensar en mantener el código desacoplado y cohesivo.

Segundo, considere esto como una manera de desarrollar su instinto. Tome nota de la situación en su cuaderno de bitácora de ingeniería: las opciones que tiene y algunas conjeturas sobre el cambio. Deje una etiqueta en el código fuente. Después, más adelante, cuando este código tenga que cambiar, podrá echar la vista atrás y ofrecerse *feedback* a sí mismo. Puede

que le ayude la próxima vez que se encuentre con una bifurcación similar en el camino.

El resto de las secciones de este capítulo contienen ideas específicas sobre diseño, pero todas están motivadas por este principio.

Las secciones relacionadas incluyen

- Tema 9, “DRY: los males de la duplicación”.
- Tema 10, “Ortogonalidad”.
- Tema 11, “Reversibilidad”.
- Tema 14, “Lenguajes de dominio”.
- Tema 28, “Desacoplamiento”.
- Tema 30, “Transformar la programación”.
- Tema 31, “Impuesto sobre la herencia”.

Retos

- Piense en un principio de diseño que utilice con regularidad. ¿Está pensado para hacer que las cosas sean fáciles de cambiar?
- Piense también en lenguajes y paradigmas de programación (programación orientada a objetos, programación funcional, programación reactiva, etc.). ¿Alguno de ellos tiene grandes aspectos positivos o grandes aspectos negativos a la hora de ayudarle a escribir código ETC? ¿Alguno tiene las dos cosas?
  Al escribir código, ¿qué puede hacer para eliminar los aspectos negativos y enfatizar los positivos?[1]
- Muchos editores tienen soporte (ya sea integrado o mediante extensiones) para ejecutar comandos cuando se guarda un archivo. Haga que su editor muestre un mensaje “¿ETC?” cada vez que guarde[2] y utilícelo como señal para pensar en el código que acaba de escribir. ¿Es fácil de cambiar?

9 DRY: los males de la duplicación

Dar a un ordenador dos informaciones contradictorias era la manera preferida del capitán James T. Kirk de deshabilitar una inteligencia artificial acechante. Por desgracia, ese mismo principio puede resultar efectivo para hacer polvo nuestro código.

Como programadores, recopilamos, organizamos, mantenemos y aprovechamos conocimientos. Documentamos el conocimiento en especificaciones, le damos vida en código en ejecución y lo utilizamos para ofrecer las comprobaciones necesarias durante las pruebas.

Por desgracia, el conocimiento no es estable. Cambia y, a menudo, lo hace con rapidez. Nuestra comprensión de un requisito puede cambiar después de una reunión con el cliente. El gobierno cambia una regulación y una lógica de negocio se queda desfasada. Puede que las pruebas muestren que el algoritmo elegido no va a funcionar. Toda esta inestabilidad significa que pasamos gran parte de nuestro tiempo en modo de mantenimiento, reorganizando y reformulando la información en nuestros sistemas.

La mayoría de la gente asume que el mantenimiento comienza cuando se lanza una aplicación, que el mantenimiento significa arreglar fallos y mejorar funcionalidades. Creemos que esa gente se equivoca. Los programadores están constantemente en modo de mantenimiento. Nuestro entendimiento cambia día a día. Llegan requisitos nuevos, y los requisitos existentes evolucionan a medida que vamos avanzando en el proyecto. Quizá cambie el entorno. Sea cual sea la razón, el mantenimiento no es una actividad discontinua, sino una parte rutinaria del proceso de desarrollo completo.

Cuando llevamos a cabo el mantenimiento, tenemos que encontrar y cambiar las representaciones de las cosas, esas cápsulas de información incrustadas en la aplicación. El problema es que es fácil duplicar la información en las especificaciones, procesos y programas que desarrollamos y, cuando lo hacemos, damos lugar a una pesadilla de mantenimiento, una que empieza mucho antes de que se envíe la aplicación.

Nos parece que la única manera de desarrollar software de manera fiable, y de hacer que nuestros desarrollos sean más fáciles de entender y mantener, es seguir lo que llamamos principio DRY.

Cualquier porción de información debe tener una representación única, inequívoca y autoritativa dentro de un sistema.

¿Por qué lo llamamos DRY?

Truco 15. DRY: *Don’t Repeat Yourself*. No se repita.

La alternativa es tener la misma cosa expresada en dos o más lugares. Si cambia una, tiene que recordar cambiar las otras o, como ocurría con los ordenadores alienígenas, el programa acabará derrotado por una contradicción. No es una cuestión de si se acordará: es una cuestión de cuándo se le olvidará.

Verá que el principio DRY aparece una y otra vez a lo largo de este libro, a menudo en contextos que no tienen nada que ver con el código. Consideramos que es una de las herramientas más importantes del kit del programador pragmático.

En esta sección, esbozaremos los problemas de duplicación y sugeriremos estrategias para enfrentarnos a ellos.

DRY es más que código

En primer lugar, vamos a aclarar algo. En la primera edición de este libro, no hicimos un buen trabajo al explicar a qué nos referíamos con “no se repita”. Muchas personas entendieron que nos referíamos solo al código: pensaban que DRY quería decir “no copie y pegue líneas del código fuente”.

Eso es parte del principio DRY, pero es una parte pequeña y bastante trivial.

DRY tiene que ver con la duplicación de la información, del propósito. Tiene que ver con expresar la misma cosa en dos lugares diferentes, posiblemente de dos maneras distintas por completo.

Y aquí viene la prueba de fuego: cuando tiene que cambiar un solo aspecto del código, ¿se descubre a sí mismo teniendo que hacer ese cambio en múltiples lugares y en múltiples formatos diferentes? ¿Tiene que cambiar el código y la documentación, o un esquema de base de datos y una estructura que lo alberga, o...? Si es así, su código no es DRY. Así pues, vamos a ver algunos ejemplos típicos de duplicación.

Duplicación en el código

Puede que sea trivial, pero la duplicación es muy común. Veamos un ejemplo:

```
def print_balance(account)
   printf “Debits: %10.2f\n”, account.debits
   printf “Credits: %10.2f\n”, account.credits
   if account.fees < 0
      printf “Fees: %10.2f-\n”, -account.fees
   else
      printf “Fees: %10.2f\n”, account.fees
   end
printf “ ———-\n”
      if account.balance < 0
      printf “Balance: %10.2f-\n”, -account.balance
   else
      printf “Balance: %10.2f\n”, account.balance
   end
end
```

Por ahora, ignore la implicación de que estamos cometiendo el error de novato de guardar las monedas en flotantes. En vez de eso, intente detectar duplicaciones en este código. (Nosotros hemos encontrado al menos tres cosas, pero puede que usted vea más).

¿Qué ha encontrado? Esta es nuestra lista.

En primer lugar, hay una duplicación clara de tipo corta y pega de la gestión de los números negativos. Podemos arreglarlo añadiendo otra función:

```
def format_amount(value)
   result = sprintf(“%10.2f”, value.abs)
   if value < 0
      result + “-”
   else
      result + “ “
   end
end

def print_balance(account)
   printf “Debits: %10.2f\n”, account.debits
   printf “Credits: %10.2f\n”, account.credits
   printf “Fees: %s\n”, format_amount(account.fees)
   printf “ ———-\n”
   printf “Balance: %s\n”, format_amount(account.balance)
end
```

Otra duplicación es la repetición de la anchura de campo en todas las llamadas `printf`. Podríamos arreglar esto introduciendo una constante y pasándola a cada llamada, pero ¿por qué no utilizar la función existente, sin más?

```
def format_amount(value)
   result = sprintf(“%10.2f”, value.abs)
   if value < 0
      result + “-”
   else
      result + “ “
   end
end

def print_balance(account)
   printf “Debits: %s\n”, format_amount(account.debits)
   printf “Credits: %s\n”, format_amount(account.credits)
   printf “Fees: %s\n”, format_amount(account.fees)
   printf “ ———-\n”
   printf “Balance: %s\n”, format_amount(account.balance)
end
```

¿Algo más? Bueno, ¿qué pasa si el cliente pide un espacio extra entre las etiquetas y los números? Tendríamos que cambiar cinco líneas. Vamos a eliminar esa duplicación:

```
def format_amount(value)
   result = sprintf(“%10.2f”, value.abs)
   if value < 0
      result + “-”
   else
      result + “ “
   end
end

def print_line(label, value)
   printf “%-9s%s\n”, label, value
end

def report_line(label, amount)
   print_line(label + “:”, format_amount(amount))
end

def print_balance(account)
   report_line(“Debits”, account.debits)
   report_line(“Credits”, account.credits)
```

```
   report_line(“Fees”, account.fees)
   print_line(“”, “———-”)
   report_line(“Balance”, account.balance)
end
```

Si tenemos que cambiar el formato de las cantidades, cambiamos `format_amount`. Si queremos cambiar el formato de las etiquetas, cambiamos `report_line`.

Sigue habiendo una violación del principio DRY implícita: el número de guiones en la línea de separación está relacionado con la anchura del campo de las cantidades. Pero no es una correspondencia exacta: en este momento es un carácter más corta, así que cualquier signo menos siguiente se extiende más allá de la columna. Ese es el propósito del cliente y es un propósito diferente a la aplicación real de formato a las cantidades.

No todas las duplicaciones de código son duplicaciones de información

Como parte de su aplicación para pedir vino en línea, está recogiendo y validando la edad de su usuario, junto con la cantidad que está pidiendo. Según el propietario del sitio, las dos cosas deberían ser números, y ambos mayores que cero. Así pues, crea código para las validaciones:

```
def validate_age(value):
   validate_type(value, :integer)
   validate_min_integer(value, 0)
def validate_quantity(value):
   validate_type(value, :integer)
   validate_min_integer(value, 0)
```

Durante la revisión del código, el listillo residente devuelve el código, afirmando que es una violación del principio DRY: los cuerpos de las dos funciones son iguales.

Se equivoca. El código es el mismo, pero la información que representa es diferente. Las dos funciones validan dos cosas separadas que resulta que tienen las mismas reglas. Eso es una coincidencia, no una duplicación.

Duplicación en la documentación

De algún modo, se ha creado el mito de que deberíamos comentar todas las funciones. Quienes creen en esa locura producen algo como esto:

```
# Calcular las tasas para esta cuenta.
#
# * Cada cheque devuelto cuesta 20 dólares
# * Si la cuenta está en descubierto más de 3 días,
# se cobran 10 dólares por cada día
# * Si el saldo medio es superior a 2.000 dólares
# se reducen las tasas en un 50 %
def fees(a)
   f = 0
   if a.returned_check_count > 0
      f += 20 * a.returned_check_count
   end
   if a.overdraft_days > 3
      f += 10*a.overdraft_days
   end
   if a.average_balance > 2_000
      f /= 2
   end
   f
end
```

El propósito de esta función se proporciona dos veces: una vez en el comentario y otra vez en el código. El cliente cambia una tasa y tenemos que actualizar las dos cosas. Con el tiempo, podemos garantizar casi con total seguridad que el comentario y el código dejarán de ir a la par.

Pregúntese qué aporta el comentario al código. Desde nuestro punto de vista, simplemente es una manera de compensar el uso de nombres y disposiciones malos. ¿Qué tal esto?:

```
def calculate_account_fees(account)
   fees = 20 * account.returned_check_count
   fees += 10 * account.overdraft_days if account.overdraft_days > 3
   fees /= 2 if account.average_balance > 2_000
   fees
end
```

El nombre dice lo que hace y, si alguien necesita detalles, están expuestos en la fuente. ¡Eso es DRY!

Violaciones de DRY en los datos

Nuestras estructuras de datos representan información y pueden tener problemas con el principio DRY. Vamos a echar un vistazo a una clase que representa una línea:

```
class Line {
   Point start;
   Point end;
   double length;
};
```

A primera vista, esta clase podría parecer razonable. Una línea tiene, claramente, un principio y un final, y siempre tendrá una longitud (incluso aunque sea cero). Pero tenemos duplicación. La longitud está definida por los puntos de inicio y final: cambie uno de esos puntos y la longitud cambiará. Es mejor hacer que la longitud sea un campo calculado:

```
class Line {
   Point start;
   Point end;
   double length() { return start.distanceTo(end); }
};
```

Más adelante en el proceso de desarrollo, puede que elija violar el principio DRY por razones de rendimiento. Con frecuencia, esto ocurre cuando necesita almacenar datos para evitar repetir operaciones caras. El truco es limitar el impacto. La violación no está expuesta al mundo exterior: solo los métodos dentro de la clase tienen que preocuparse por mantener las cosas claras:

```
class Line {
   private double length;
   private Point start;
   private Point end;

   public Line(Point start, Point end) {
      this.start = start;
      this.end = end;
      calculateLength();
}

   // público
   void setStart(Point p) { this.start = p; calculateLength(); }
   void setEnd(Point p) { this.end = p; calculateLength(); }
```

```
   Point getStart() { return start; }
   Point getEnd() { return end; }

   double getLength() { return length; }

   private void calculateLength() {
      this.length = start.distanceTo(end);
   }
};
```

Este ejemplo también ilustra una cuestión importante: cada vez que un módulo expone una estructura de datos, estamos acoplando todo el código que usa esa estructura a la implementación de ese módulo. Cuando sea posible, use siempre funciones de acceso para leer y escribir los atributos de objetos. Eso hará que sea más fácil añadir funcionalidad en el futuro. Este uso de funciones de acceso está conectado con el principio de acceso uniforme de Meyer, descrito en *Construcción de software orientado a objetos* [Mey97], que afirma que:

> *Todos los servicios ofrecidos por un módulo deberían estar disponibles mediante una notación uniforme, que no revela si se implementan a través del almacenamiento o a través de la computación.*

Duplicación relativa a la representación

Nuestro código se conecta con el mundo exterior: otras bibliotecas mediante API, otros servicios mediante llamadas remotas, datos en fuentes externas, etc. Y, prácticamente, cada vez que eso ocurre, introducimos algún tipo de violación de DRY: su código debe tener información que también esté presente en el elemento externo. Necesita conocer la API, el esquema, el significado de los códigos de error o lo que sea. Aquí la duplicación es que dos cosas (su código y la entidad externa) deben tener conocimiento de la representación de su interrelación. Haga un cambio en una de las dos cosas y la otra se estropeará.

Esta duplicación es inevitable, pero puede atenuarse. Veamos algunas estrategias.

Duplicación entre API internas

Para las API internas, busque herramientas que le permitan especificar la API en algún tipo de formato neutral. Por lo general, estas herramientas generarán documentación, API simuladas, pruebas funcionales y clientes de API, estos últimos en varios lenguajes diferentes. Idealmente, la herramienta almacenará todas sus API en un repositorio central, permitiendo que se compartan entre los equipos.

Duplicación entre API externas

Cada vez con más frecuencia, verá que las API públicas se documentan de manera formal mediante el uso de algo como OpenAPI.[3] Esto le permite importar la especificación de la API en sus herramientas de API locales e integrarla de modo más fiable con el servicio.

Si no puede encontrar una especificación así, plantéese crear una y publicarla. No solo otras personas la encontrarán útil, sino que quizá incluso reciba ayuda para mantenerla.

Duplicación con fuentes de datos

Muchas fuentes de datos le permiten explorar su esquema de datos. Esto puede usarse para eliminar gran parte de la duplicación entre ellas y su código. En vez de crear a mano el código para contener los datos almacenados, puede generar los contenedores directamente desde el esquema. Muchos *frameworks* de persistencia harán ese trabajo por usted.

Hay otra opción, que es la que a menudo preferimos nosotros. En vez de escribir código que represente datos externos en una estructura fija (una instancia de un *struct* o una clase, por ejemplo), simplemente métalo en una estructura de datos clave-valor (puede que su lenguaje la llame mapa, *hash*, diccionario o incluso objeto).

Por sí solo, esto es arriesgado: se pierde gran parte de la seguridad de saber con qué datos estamos trabajando. Por tanto, recomendamos añadir una segunda capa a esta solución: una simple *suite* de validación basada en tablas que verifique que el mapa que ha creado contiene al menos los datos que necesita, en el formato en el que los necesita. Es posible que su herramienta de documentación de API pueda generar esto.

Duplicación entre varios desarrolladores

Quizá el tipo de duplicación más difícil de detectar y gestionar sea el que se produce entre diferentes desarrolladores en un proyecto. Conjuntos enteros de funcionalidades pueden duplicarse sin que nos demos cuenta y esa duplicación puede pasar inadvertida durante años, lo que lleva a problemas de mantenimiento. Hemos oído hablar de primera mano de un estado de EE. UU. cuyos sistemas informáticos gubernamentales se inspeccionaron para asegurarse de que estaban preparados para enfrentarse al efecto 2000. La auditoría reveló más de 10.000 programas que contenían cada uno una versión diferente del código de validación de números de la Seguridad Social.

En un nivel alto, enfréntese al problema creando un equipo fuerte y muy unido con buenas comunicaciones.

Sin embargo, a nivel de módulos, el problema es más traicionero. Los datos o las funcionalidades que se necesitan de forma común y que no entran en un área de responsabilidad obvia pueden implementarse muchas veces.

Consideramos que la mejor manera de lidiar con esto es incentivar la comunicación activa y frecuente entre desarrolladores.

Quizá puedan celebrarse reuniones de pie *scrum* diarias. Configure foros (como los canales Slack) para hablar sobre problemas comunes. Esto ofrece una manera no intrusiva de comunicarse (a veces a través de múltiples sitios) mientras se conserva un historial permanente de todo lo que se dice.

Nombre a un miembro del equipo bibliotecario del proyecto, que tenga como tarea facilitar el intercambio de conocimientos. Tenga un lugar central en el árbol fuente en el que puedan depositarse los *scripts* y las rutinas de servicio, y asegúrese de leer el código fuente y la documentación de otras personas, ya sea de manera informal o durante las revisiones de código. No está cotilleando, está aprendiendo de ellas. Y, recuerde, el acceso es recíproco; no se enfade tampoco si otras personas leen con atención (¿o de forma descuidada?) su código.

Truco 16. Facilite la reutilización.

Lo que está intentando hacer es fomentar un entorno en el que sea más fácil encontrar y reutilizar cosas existentes que escribirlas usted mismo. Si no es fácil, la gente no lo hará. Y, si no logra reutilizarlo, se arriesga a duplicar la información.

Las secciones relacionadas incluyen

- Tema 8, “La esencia del buen diseño”.
- Tema 28, “Desacoplamiento”.
- Tema 32, “Configuración”.
- Tema 38, “Programar por casualidad”.
- Tema 40, “Refactorización”.

10 Ortogonalidad

La ortogonalidad es un concepto crítico si quiere producir sistemas que sean fáciles de diseñar, construir, probar y ampliar. Sin embargo, este concepto rara vez se enseña de manera directa. A menudo, es una característica implícita de otros métodos y técnicas que aprendemos. Eso es un error. Una vez que aprenda a aplicar el principio de ortogonalidad de modo directo, observará una mejora inmediata en la calidad de los sistemas que produce.

¿Qué es la ortogonalidad?

“Ortogonalidad” es un término que se toma prestado de la geometría. Dos líneas son ortogonales si se encuentran en un ángulo recto, como los ejes de un gráfico. En lo relativo a los vectores, las dos líneas son independientes. Cuando el número 1 del diagrama (figura 2.1) se mueve hacia el norte, no afecta a cuánto está hacia el este o el oeste. El número 2 se mueve hacia el este, pero no hacia el norte o el sur.

Figura 2.1.

En informática, el término ha acabado significando una especie de independencia o desacoplamiento. Dos o más cosas son ortogonales si los cambios en una no afectan a ninguna de las demás. En un sistema bien diseñado, el código de la base de datos será ortogonal respecto a la interfaz de usuario: puede cambiar la interfaz sin afectar a la base de datos, y modificar las bases de datos sin cambiar la interfaz. Antes de hablar de los beneficios de los sistemas ortogonales, echemos un vistazo a un sistema que no lo es.

Un sistema no ortogonal

Va haciendo una excursión en el helicóptero por el Gran Cañón cuando el piloto, que ha cometido el error evidente de tomar pescado para comer, de pronto gime y se desmaya. Por fortuna, le ha dejado volando a 30 metros de altura.

Por un golpe de suerte, anoche leyó un artículo de Wikipedia sobre helicópteros. Sabe que los helicópteros tienen cuatro controles básicos. El cíclico es el mando que sujeta con la mano derecha. Si la mueve, el helicóptero se mueve en la dirección correspondiente. Con la mano izquierda, sujeta la palanca del paso colectivo. Levántela y aumentará el ángulo en todas las aspas, generando la elevación. En el extremo de la palanca del mando colectivo está la palanca de gases. Por último, tenemos

dos pedales, que varían la cantidad de impulso del rotor de cola y ayudan al helicóptero a girar.

"¡Fácil!", piensa. "Si bajas la palanca del paso colectivo y desciendes con elegancia hasta el suelo, eres un héroe". Sin embargo, cuando lo intenta, descubre que la vida no es tan fácil. El morro del helicóptero baja y el aparato empieza a bajar girando en una espiral hacia la izquierda. De pronto se da cuenta de que está manejando un sistema en el que cualquier acción sobre un control tiene efectos secundarios. Si baja la palanca de la izquierda tendrá que añadir un movimiento de compensación hacia atrás con la palanca de la derecha y pisar el pedal derecho. Pero cada uno de estos cambios afecta a todos los demás controles otra vez. De repente, está haciendo malabares con un sistema increíblemente complejo, donde cada cambio causa un impacto sobre todas las demás acciones. Su carga de trabajo es enorme: sus manos y sus pies están en movimiento todo el tiempo, intentando equilibrar todas las fuerzas que están interactuando. Desde luego, los mandos del helicóptero no son ortogonales.

Beneficios de la ortogonalidad

Como ilustra el ejemplo del helicóptero, los sistemas no ortogonales son, de manera inherente, más difíciles de cambiar y controlar. Cuando los componentes de un sistema son muy interdependientes, no existen los arreglos locales.

Truco 17. Elimine efectos entre cosas no relacionadas.

Queremos diseñar componentes que sean autónomos: independientes y con un solo propósito bien definido (lo que Yourdon y Constantine llaman "cohesión" en *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* [YC79]). Cuando los componentes se aíslan unos de otros, sabe que puede modificar uno sin tener que preocuparse por los demás. Siempre y cuando no cambie las interfaces externas de ese componente, puede estar seguro de que no va a causar problemas que se propaguen por todo el sistema.

Si escribe sistemas ortogonales, conseguirá dos beneficios importantes: un aumento de la productividad y una reducción del riesgo.

Incrementar la productividad

- Los cambios están localizados, así que el tiempo de desarrollo y el tiempo de pruebas se reducen. Es más fácil escribir componentes relativamente pequeños e independientes que un solo bloque grande de código. Los componentes simples pueden diseñarse, convertirse en código, probarse y, después, olvidarse; no hay necesidad de seguir cambiando código existente a medida que se añade código nuevo.
- Un enfoque ortogonal también fomenta la reutilización. Si los componentes tienen responsabilidades específicas bien definidas, pueden combinarse con componentes nuevos de maneras que sus implementadores originales no habían previsto. Cuanto más débil sea el acoplamiento de sus sistemas, más fácil será reconfigurarlos y rediseñarlos.
- Hay una ganancia bastante sutil en la productividad cuando se combinan componentes ortogonales. Supongamos que un componente hace M cosas distintas y otro hace N cosas. Si son ortogonales y los combina, el resultado hará M × N cosas. Sin embargo, si los dos componentes no son ortogonales, habrá un solapamiento, y el resultado hará menos. Conseguirá más funcionalidad por esfuerzo unitario si combina componentes ortogonales.

Reducir el riesgo

Un enfoque ortogonal reduce los riesgos inherentes en cualquier desarrollo.

- Las secciones enfermas de código se aíslan. Si un módulo está enfermo, es menos probable que se propaguen los síntomas por el resto del sistema. También es más fácil extirparlo y trasplantar algo nuevo y sano.

- El sistema resultante es menos frágil. Si hace cualquier cambio o arreglo en un área concreta, cualquier problema que genere quedará restringido a esa área.
- Es probable que un sistema ortogonal se pruebe mejor, porque será más fácil diseñar y ejecutar pruebas en sus componentes.
- No estará tan atado a un proveedor, producto o plataforma en particular, porque las interfaces con estos componentes de tercero estarán aisladas en partes más pequeñas del desarrollo general.

Echemos un vistazo a algunas de las formas en que podemos aplicar el principio de ortogonalidad a nuestro trabajo.

Diseño

La mayoría de los desarrolladores están familiarizados con la necesidad de diseñar sistemas ortogonales, aunque puede que usen palabras como “modular”, “basado en componentes” y “con capas” para describir el proceso. Los sistemas deberían estar compuestos de un conjunto de módulos que cooperen, cada uno de los cuales implementa funcionalidad independiente de los demás. A veces, estos componentes se organizan en capas, cada una de las cuales proporciona un nivel de abstracción. Este enfoque en capas es una manera potente de diseñar sistemas ortogonales. Puesto que cada capa usa solo las abstracciones proporcionadas por las capas que hay debajo de ella, tenemos gran flexibilidad a la hora de cambiar las implementaciones subyacentes sin afectar al código. El uso de capas también reduce el riesgo de dependencias desmedidas entre módulos. A menudo, verá el uso de capas expresado en diagramas (figura 2.2):

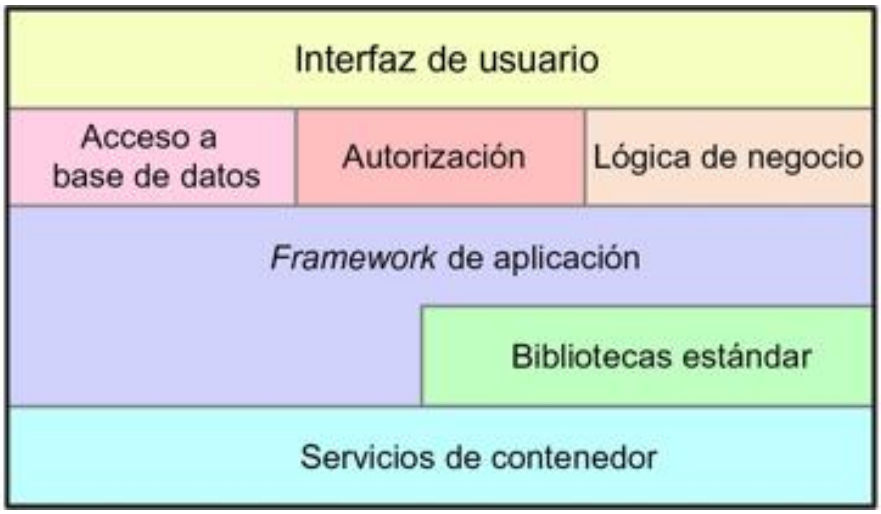


Figura 2.2.

Hay una prueba sencilla para el diseño ortogonal. Una vez que tenga sus componentes planeados, pregúntese: “Si cambio de forma drástica los requisitos que hay tras una función particular, ¿cuántos módulos se ven afectados?”. En un sistema ortogonal, la respuesta debería ser “uno”.[4] Mover un botón en el panel de una GUI no debería requerir un cambio en el esquema de la base de datos. Añadir ayuda dependiente del contexto no debería cambiar el subsistema de facturación.

Vamos a pensar en un sistema complejo para la monitorización y el control de una central térmica. El requisito original exigía una interfaz gráfica de usuario, pero se cambiaron los requisitos para añadir una interfaz móvil que permitiese a los ingenieros monitorizar valores clave. En un sistema con diseño ortogonal, habría que cambiar solo aquellos módulos asociados a la interfaz de usuario para ocuparse de esto: la lógica subyacente del control de la central permanecería intacta. De hecho, si estructuramos el sistema con cuidado, deberíamos poder soportar ambas interfaces con la misma base de código subyacente.

Pregúntese también lo desacoplado que está su diseño de los cambios en el mundo real. ¿Utiliza un número de teléfono como identificador del cliente? ¿Qué ocurre cuando la compañía telefónica reasigna los prefijos? Los códigos postales, los números de la seguridad social o de identificación del gobierno, las direcciones de correo electrónico y los dominios son identificadores externos sobre los que usted no tiene ningún control y que

podrían cambiar en cualquier momento por cualquier razón. No dependa de las propiedades de cosas que no puede controlar.

Herramientas y bibliotecas

Ponga cuidado en preservar la ortogonalidad de su sistema cuando introduzca herramientas y bibliotecas de terceros. Elija sus tecnologías con inteligencia.

Cuando introduzca un juego de herramientas (o incluso una biblioteca de otros miembros de su equipo), pregúntese si impone cambios en su código que no deberían estar ahí. Si un esquema de persistencia de objetos es transparente, entonces es ortogonal. Si requiere que cree o acceda a objetos de una manera especial, entonces no lo es. Mantener esos detalles aislados de su código tiene la ventaja añadida de que hace que sea más fácil cambiar de proveedor en el futuro.

El sistema Enterprise JavaBeans (EJB) es un ejemplo interesante de ortogonalidad. En la mayoría de sistemas orientados a las transacciones, el código de aplicación tiene que definir el inicio y el final de cada transacción. Con EJB, esta información se expresa de manera declarativa como anotaciones, fuera de los métodos que hacen el trabajo. El mismo código de aplicación puede ejecutarse en entornos de transacción EJB diferentes sin cambios.

En cierto modo, EJB es un ejemplo de patrón Decorator: añadir funcionalidad a las cosas sin cambiarlas. Este estilo de programación puede usarse en casi cualquier lenguaje de programación y no requiere necesariamente un *framework* o una biblioteca. Solo hace falta un poco de disciplina al programar.

Creación de código

Cada vez que escribe código, corre el riesgo de reducir la ortogonalidad de su aplicación. A menos que monitorice todo el tiempo no solo lo que está haciendo, sino también el contexto más grande de la aplicación, puede que duplique sin querer la funcionalidad en algún otro módulo o que exprese información existente dos veces.

Hay muchas técnicas que puede utilizar para mantener la ortogonalidad:

- **Mantenga el código desacoplado:** Escriba código tímido, módulos que no revelen nada innecesario a otros módulos y que no dependan de las implementaciones de otros módulos. Pruebe a utilizar la ley de Demeter, que veremos en el tema 28, “Desacoplamiento”. Si necesita cambiar el estado de un objeto, haga que el objeto se ocupe por usted. De ese modo, su código permanece aislado de la implementación del otro código y aumenta las probabilidades de seguir siendo ortogonal.
- **Evite los datos globales:** Cada vez que su código hace referencia a datos globales, se vincula a los otros componentes que comparten esos datos. Incluso datos globales que solo planea leer pueden generar problemas (por ejemplo, si de repente necesita cambiar el código para que sea multihilo). En general, su código será más fácil de entender y mantener si pasa de manera explícita cualquier contexto a sus módulos. En las aplicaciones orientadas a objetos, el contexto se pasa a menudo como parámetros a constructores de objetos. En otro código, puede crear estructuras que contengan el contexto y e ir pasando referencias a ellas. El patrón Singleton en *Patrones de Diseño. Elementos de software orientado a objetos reutilizable* [GHJV95] es una manera de garantizar que solo hay una instancia de un objeto de una clase en particular. Mucha gente usa estos objetos Singleton como una especie de variable global (sobre todo en lenguajes, como Java, que no son compatibles con el concepto de globales). Tenga cuidado con los Singleton; también pueden conducir a enlaces innecesarios.
- **Evite funciones similares:** A menudo, se topará con un conjunto de funciones que parecen similares; puede que compartan código común al principio y al final, pero cada una tiene un algoritmo central diferente. El código duplicado es síntoma de un problema estructural. Eche un vistazo al patrón Strategy en *Patrones de diseño* para ver una implementación mejor.

Acostúmbrese a ser crítico con su código todo el tiempo. Busque oportunidades para reorganizarlo para mejorar su estructura y su ortogonalidad. Este proceso se llama “refactorización”, y es tan importante

que hemos dedicado una sección a hablar de él (consulte el tema 40, “Refactorización”).

Pruebas

Un sistema diseñado e implementado de manera ortogonal es más fácil de probar. Puesto que las interacciones entre los componentes del sistema están formalizadas y limitadas, hay más pruebas del sistema que pueden realizarse a nivel de módulos individuales. Esto es bueno, porque las pruebas a nivel de módulo (o unitarias) son bastante más fáciles de especificar y realizar que las pruebas de integración. De hecho, sugerimos que estas pruebas se realicen de forma automática como parte del proceso de construcción habitual (consulte el tema 41, “Probar para escribir código”).

Escribir pruebas unitarias es en sí mismo una prueba interesante de ortogonalidad. ¿Qué hace falta para que una prueba unitaria se construya y se ejecute? ¿Hay que importar un porcentaje grande del resto del código del sistema? Si es así, ha encontrado un módulo que no está bien desacoplado del resto del sistema.

La reparación de fallos también es un buen momento para evaluar la ortogonalidad del sistema en su conjunto. Cuando se tope con un problema, valore lo localizada que está la solución. ¿Modifica solo un módulo o los cambios están desperdigados por todo el sistema? Cuando hace un cambio, ¿lo arregla todo o misteriosamente surgen otros problemas? Esta es una buena oportunidad para aplicar la automatización. Si utiliza un sistema de control de versiones (cosa que hará después de leer el tema 19, “Control de versiones”), etiquete los arreglos de los fallos cuando devuelva el código al repositorio después de probarlo. Después, puede realizar informes mensuales analizando las tendencias en el número de archivos fuente afectados por cada arreglo de fallos.

Documentación

Aunque quizá resulte sorprendente, la ortogonalidad también se aplica a la documentación. Los ejes son el contenido y la presentación. Con una

documentación ortogonal de verdad, debería ser capaz de cambiar la apariencia de forma radical sin cambiar el contenido. Los procesadores de texto ofrecen hojas de estilo y macros que ayudan. Nosotros, personalmente, preferimos utilizar un sistema de marcado como Markdown: cuando escribimos, nos centramos solo en el contenido y dejamos la presentación en manos de la herramienta que utilicemos para plasmarla.[5]

Vivir con ortogonalidad

La ortogonalidad está muy relacionada con el principio DRY. Con DRY, se intenta minimizar la duplicación dentro de un sistema, mientras que con la ortogonalidad se reduce la interdependencia entre los componentes del sistema. Puede que sea una palabra torpe, pero, si utiliza el principio de ortogonalidad, combinado estrechamente con el principio DRY, descubrirá que los sistemas que desarrolla son más flexibles, más comprensibles y más fáciles de depurar, probar y mantener.

Si pasa a formar parte de un proyecto en el que la gente está desesperada por los problemas para hacer cambios y donde cada cambio parece hacer que otras cuatro cosas se estropeen, recuerde la pesadilla del helicóptero. Es probable que el proyecto no esté diseñado ni el código esté creado de manera ortogonal. Es hora de refactorizar.

Y, si es piloto de helicóptero, no se coma el pescado...

Las secciones relacionadas incluyen

- Tema 3, “Entropía del software”.
- Tema 8, “La esencia del buen diseño”.
- Tema 11, “Reversibilidad”.
- Tema 28, “Desacoplamiento”.
- Tema 31, “Impuesto sobre la herencia”.
- Tema 33, “Romper el acoplamiento temporal”.
- Tema 34, “Estado compartido es estado incorrecto”.
- Tema 36, “Pizarras”.

Retos

- Piense en la diferencia entre herramientas que tienen una interfaz gráfica de usuario y servicios de línea de comandos pequeños pero combinables utilizados en intérpretes de comandos. ¿Qué conjunto es más ortogonal y por qué? ¿Cuál es más fácil de utilizar exactamente para el propósito para el que está pensado? ¿Qué conjunto es más fácil de combinar con otras herramientas para afrontar retos nuevos? ¿Qué conjunto es más fácil de aprender?
- C++ soporta herencia múltiple y Java permite que una clase implemente interfaces múltiples. Ruby tiene *mixins*. ¿Qué impacto tiene sobre la ortogonalidad utilizar estas características? ¿Hay alguna diferencia en el impacto entre utilizar herencia múltiple e interfaces múltiples? ¿Hay alguna diferencia entre usar delegación y usar herencia?

Ejercicios

Ejercicio 1

Le piden que lea un archivo línea por línea. Para cada línea, debe dividirla en campos. ¿Cuál de los siguientes conjuntos de definiciones de pseudoclase tiene más probabilidades de ser ortogonal?

```
class Split1 {
   constructor(fileName) # abre el archivo para lectura
   def readNextLine() # pasa a la siguiente línea
   def getField(n) # devuelve el enésimo campo en la línea actual
}
```

o

```
class Split2 {
   constructor(line) # divide una línea
   def getField(n) # devuelve el enésimo campo en la línea actual
}
```

Ejercicio 2

¿Cuáles son las diferencias en la ortogonalidad entre lenguajes orientados a objetos y lenguajes funcionales? ¿Son estas diferencias inherentes a los lenguajes en sí o solo a la manera en que la gente los usa?

11 Reversibilidad

Nada es más peligroso que una idea si es la única que tienes.

—Émile-Auguste Chartier (Alain), Divagaciones sobre la religión, 1938.

Los ingenieros prefieren soluciones simples y singulares a los problemas. Las pruebas matemáticas que permiten afirmar con gran confianza que `x = 2` son mucho más cómodas que los ensayos difusos acerca de la miríada de razones que dieron lugar a la Revolución Francesa. Los directores tienden a estar de acuerdo con los ingenieros: las respuestas fáciles y singulares se adaptan mejor a hojas de cálculo y planes de proyecto.

¡Ojalá el mundo real cooperase! Por desgracia, aunque `x` es `2` hoy, puede que necesite ser `5` mañana, y `3` la semana que viene. Nada es para siempre y, si dependemos mucho de algún hecho, podemos estar casi seguros de que cambiará. Siempre hay más de una manera de implementar algo y suele haber más de un proveedor disponible para ofrecer un producto de terceros. Si aborda un proyecto obstaculizado por la noción miope de que solo hay una forma de hacerlo, puede que se encuentre con una desagradable sorpresa. Muchos equipos de proyectos aprenden por las malas a medida que el futuro se abre ante ellos:

—¡Pero dijiste que usaríamos la base de datos XYZ! ¡Hemos escrito el 85 % del código del proyecto, no podemos cambiarlo ahora! —protestó el programador.
—Lo siento, pero nuestra empresa ha decidido estandarizar con la base de datos PDQ en su lugar, para todos los proyectos. No está en mis manos. Tendremos que rehacer el código. Todos trabajaréis los fines de semana hasta que os digamos.

Los cambios no tienen por qué ser draconianos, ni siquiera inmediatos. Pero, a medida que pase el tiempo y progrese el proyecto, se encontrará atascado en una posición insostenible. Con cada decisión crítica, el equipo del proyecto se compromete con una diana más pequeña, una versión más estrecha de la realidad que tiene menos opciones.

Para cuando se han tomado muchas decisiones críticas, la diana se vuelve tan pequeña que, si se mueve o cambia el viento o una mariposa bate las alas en Tokio, no damos en el blanco.[6] Y puede que fallemos por mucho.

El problema es que las decisiones críticas no pueden revertirse con facilidad.

Una vez que decidimos usar la base de datos de este proveedor o ese patrón arquitectónico o un modelo de despliegue determinado, nos estamos comprometiendo con un curso de acción que no puede deshacerse, salvo con un gran coste.

Reversibilidad

Muchos de los temas de este libro están dirigidos a producir software flexible y adaptable. Al ceñirnos a sus recomendaciones (sobre todo el principio DRY, el desacoplamiento y el uso de la configuración externa), no tenemos que tomar tantas decisiones críticas irreversibles. Esto es bueno, porque no siempre tomamos las mejores decisiones a la primera. Nos comprometemos con una tecnología determinada solo para descubrir que no podemos contratar a suficiente personal con las habilidades necesarias. Nos atamos a un proveedor de terceros concreto justo antes de que lo compre su competidor. Los requisitos, los usuarios y el hardware cambian más rápido de lo que nosotros podemos desarrollar el software.

Supongamos que decide, al principio del proyecto, utilizar una base de datos del proveedor A. Mucho tiempo después, durante una prueba de rendimiento, descubre que esa base de datos es demasiado lenta, sin más, pero que la base de datos documental del proveedor B es más rápida. Con los proyectos más convencionales, no tiene suerte. La mayor parte del tiempo, las llamadas a productos de terceros están enmarañadas por todo el código. Pero, si abstrajo de verdad la idea de la base de datos, hasta el punto de que simplemente proporciona persistencia como un servicio, entonces tiene flexibilidad para cambiar de caballo a mitad de camino.

De modo similar, supongamos que el proyecto empieza como una aplicación basada en navegador, pero después, en el último momento, en marketing deciden que lo que quieren de verdad es una aplicación móvil. ¿Cómo de difícil le resultaría eso? En un mundo ideal, no debería afectarle demasiado, al menos en el lado del servidor. Estaría eliminando parte de la renderización HTML y sustituyéndola por una API.

El error está en asumir que toda decisión es inamovible y en no prepararse para las contingencias que puedan surgir. En vez de grabar las decisiones en piedra, piense en escribirlas en la arena de la playa. En cualquier momento puede llegar una gran ola y borrarlas.

Truco 18. No hay decisiones finales.

Arquitectura flexible

Aunque mucha gente intenta mantener su código flexible, también hay que pensar en mantener la flexibilidad en las áreas de arquitectura, despliegue e integración de proveedores. Estamos escribiendo esto en 2019. Desde el cambio de siglo, hemos visto las siguientes arquitecturas del lado del servidor que se han llamado “mejor práctica”:

- Un gran trozo de hierro.
- Grupos de unidades centrales.
- Clústeres de equilibrio de carga de hardware genérico.
- Máquinas virtuales basadas en la nube que ejecutan aplicaciones.
- Máquinas virtuales basadas en la nube que ejecutan servicios.
- Versiones en contenedores de lo anterior.
- Aplicaciones sin servidor soportadas en la nube.
- E, inevitablemente, una vuelta aparente a los grandes trozos de hierro para algunas tareas.

Adelante, añada las últimas modas importantes a esta lista y mírela con asombro: es un milagro que cualquier cosa haya funcionado alguna vez.

¿Cómo puede planificar para tratar con este tipo de volatilidad arquitectónica? No puede.

Lo que puede hacer es que sea fácil de cambiar. Oculte las API de terceros tras sus propias capas de abstracción. Desglose su código en componentes: incluso si acaba desplegándolos en un único servidor masivo, este enfoque es mucho más fácil que tomar una aplicación monolítica y dividirla. (Tenemos cicatrices que lo demuestran).

Y, aunque esto no es específicamente un problema de reversibilidad, le daremos un último consejo.

Truco 19. Evite seguir las modas.

Nadie sabe lo que depara el futuro, ¡especialmente nosotros! Así pues, prepare su código para que siga avanzando cuando pueda y para que encaje golpes cuando deba.

Las secciones relacionadas incluyen

- Tema 8, “La esencia del buen diseño”.
- Tema 10, “Ortogonalidad”.
- Tema 19, “Control de versiones”.
- Tema 28, “Desacoplamiento”.
- Tema 45, “El pozo de los requisitos”.
- Tema 51, “Kit pragmático básico”.

Retos

- Es hora de hablar un poco de mecánica cuántica con el gato de Schrödinger.
  Imagine que tiene un gato en una caja cerrada, junto con una partícula radioactiva. La partícula tiene exactamente un 50 % de probabilidades de fisionarse en dos. Si lo hace, el gato morirá. Si no lo hace, al gato no le pasará nada. Así pues, ¿el gato está vivo o muerto? Según Schrödinger, la respuesta correcta es las dos cosas (al menos mientras la caja siga cerrada). Cada vez que se produce una reacción subnuclear que tiene dos resultados posibles, el universo se clona. En uno, el evento ocurre, y en el otro, no. El gato está vivo en un universo y muerto en el otro. Solo al abrir la caja sabe en qué universo está.
  No es de extrañar que crear código para el futuro sea difícil.
  Pero piense en la evolución del código como si fuese una caja llena de gatos de Schrödinger: cada decisión tiene como resultado una

versión diferente del futuro. ¿Cuántos futuros posibles puede soportar su código? ¿Cuáles son más probables? ¿Cómo de difícil será soportarlos cuando llegue el momento?
¿Se atreve a abrir la caja?

12 Balas trazadoras

Preparen, fuego, apunten...

—Anónimo.

A menudo pensamos en dar en una diana cuando desarrollamos software. En realidad, no estamos disparando a nada en el campo de tiro, pero aun así se trata de una metáfora muy visual y útil. En concreto, es interesante pensar en cómo dar en la diana en un mundo complejo y cambiante.

La respuesta, por supuesto, depende de la naturaleza del dispositivo con el que esté apuntando. Con muchos, solo tiene una oportunidad para apuntar y, después, ver si da en el blanco o no. Pero hay una manera mejor.

¿Ha visto todas esas películas, series y videojuegos donde la gente dispara con metralletas? En esas escenas, verá con frecuencia la trayectoria de las balas como rastros brillantes en el aire. Esos rastros vienen de las balas trazadoras.

Las balas trazadoras se cargan a intervalos junto con la munición normal. Cuando se disparan, su fósforo se prende y deja un rastro pirotécnico desde el arma hasta lo que alcancen. Si las trazadoras están dando en el blanco, entonces las balas normales también. Los soldados utilizan balas trazadoras para perfeccionar su puntería: es un *feedback* pragmático en tiempo real y en condiciones reales.

Ese mismo principio se aplica a los proyectos, en particular cuando estamos creando algo que no se ha creado antes. Utilizamos el término “desarrollo con balas trazadoras” para ilustrar de manera visual la necesidad de recibir un *feedback* inmediato en condiciones reales con un blanco móvil.

Al igual que los artilleros, estamos intentando alcanzar un objetivo a ciegas. Como nuestros usuarios nunca han visto un sistema como este antes,

sus requisitos pueden ser vagos. Como es posible que estemos usando algoritmos, técnicas, lenguajes o bibliotecas con los que no estamos familiarizados, puede que nos enfrentemos a una gran cantidad de elementos desconocidos. Y, como los proyectos tardan un tiempo en completarse, podemos estar bastante seguros de que el entorno en el que estamos trabajando cambiará antes de que hayamos acabado.

La respuesta clásica es especificar el sistema hasta la muerte. Producir resmas de papel en las que se detallen los pormenores de cada requisito, amarrando cada elemento desconocido y restringiendo el entorno. Disparar el arma usando navegación por estima. Hacer un gran cálculo por adelantado, disparar y cruzar los dedos.

Sin embargo, los programadores pragmáticos tienden a preferir el uso del equivalente en software a las balas trazadoras.

Código que brilla en la oscuridad

Las balas trazadoras funcionan porque actúan al mismo tiempo y con las mismas restricciones que las balas reales. Llegan al objetivo rápido, así que el artillero recibe un *feedback* inmediato. Y, desde un punto de vista práctico, son una solución relativamente barata.

Para lograr el mismo efecto en el código, buscamos algo que nos lleve desde un requisito a algún aspecto del sistema final de manera rápida, visible y repetible.

Busque los requisitos importantes, los que definen el sistema. Busque las áreas en las que tenga duda y donde vea los mayores riesgos. Después, priorice su desarrollo de manera que esas sean las primeras áreas para las que escriba código.

Truco 20. Use balas trazadoras para encontrar el objetivo.

De hecho, dada la complejidad de la configuración de los proyectos de hoy, con multitud de herramientas y dependencias externas, las balas trazadoras son aún más importantes. Para nosotros, la primera bala trazadora es simplemente crear el programa, añadir un “¡hola, mundo!” y asegurarnos de que se compila y se ejecuta. Después, buscamos áreas de

incertidumbre en la aplicación general y añadimos el esqueleto necesario para hacerla funcionar.

Fíjese en el siguiente diagrama (figura 2.3). Este sistema tiene cinco capas arquitectónicas. Nos preocupa un poco saber cómo se integrarían, así que buscamos una característica simple que nos deje ejercitarlas juntas. La línea diagonal busca la ruta que toma esa característica por el código. Para hacer que funcione, solo tenemos que implementar las áreas con sombreado sólido en cada capa: los garabatos serpenteantes se harán después.

Una vez nos embarcamos en un proyecto de marketing con base de datos cliente-servidor complejo. Una parte de sus requisitos era la capacidad para especificar y ejecutar consultas temporales. Los servidores era una serie de bases de datos relacionales y especializadas. La UI del cliente, escrita en un lenguaje aleatorio A, utilizaba un conjunto de bibliotecas escritas en un lenguaje diferente para proporcionar una interfaz a los servidores. La consulta del usuario se almacenaba en el servidor en una notación del estilo de Lisp antes de convertirse a SQL optimizado justo antes de la ejecución. Había muchos elementos desconocidos y muchos entornos diferentes, y nadie estaba muy seguro de cómo debería comportarse la UI.

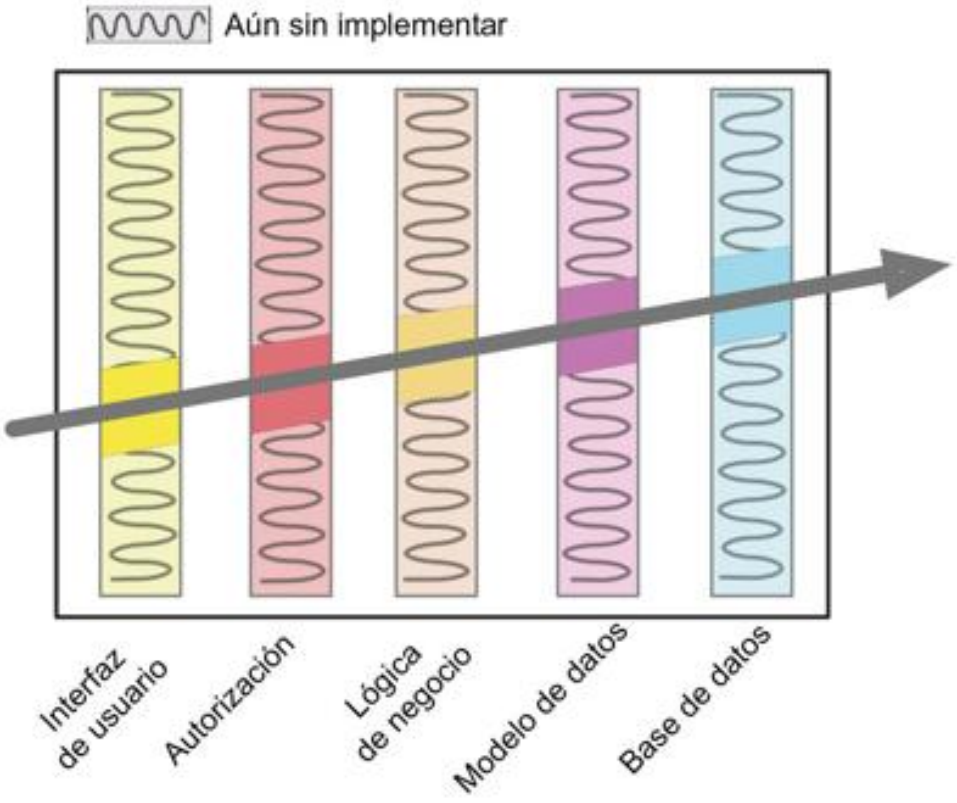


Figura 2.3.

Era una gran oportunidad para utilizar código trazador. Desarrollamos el *framework* para el *front-end*, bibliotecas para representar las consultas y una estructura para convertir una consulta almacenada en una consulta específica de una base de datos. Después, lo juntamos todo y comprobamos que funcionaba. Para la construcción inicial, lo único que podíamos hacer era enviar una consulta que hiciese una lista con todas las filas de una tabla, pero demostró que la UI podía hablar con las bibliotecas, las bibliotecas podían serializar y deserializar una consulta y el servidor podía generar SQL a partir del resultado. En los meses siguientes fuimos dando más cuerpo de manera gradual a esta estructura básica, añadiendo funcionalidad nueva aumentando cada componente del código trazador en paralelo. Cuando la UI añadía un nuevo tipo de consulta, la biblioteca crecía y la generación de SQL se volvía más sofisticada.

El código trazador no es desechable: se escribe para conservarse. Contiene todas las comprobaciones de errores, estructuraciones, documentación y autocomprobaciones que tiene cualquier porción del código de producción. Lo único que pasa es que no es totalmente funcional. Sin embargo, una vez que haya conseguido una conexión de extremo a extremo entre los componentes de su sistema, puede comprobar lo cerca que está de la diana y ajustar si es necesario. Una vez que esté en el objetivo, añadir funcionalidad será fácil.

El desarrollo trazador está en consonancia con la idea de que un proyecto nunca está acabado: siempre habrá cambios requeridos y funciones que añadir. Se trata de un enfoque gradual.

La alternativa convencional es un tipo de enfoque de ingeniería pesada: el código se divide en módulos, cuyo código se escribe en un vacío. Los módulos se combinan en subconjuntos, que se combinan a su vez hasta que, un día, tenemos una aplicación completa. Solo entonces puede presentarse al usuario y probarse la aplicación en su conjunto.

El enfoque del código trazador tiene muchas ventajas:

- **Los usuarios ven algo en funcionamiento pronto:** Si ha comunicado con éxito lo que está haciendo (consulte el tema 52, “Deleite a sus usuarios”), sus usuarios sabrán que están viendo algo

que aún no está maduro. No se sentirán decepcionados por la falta de funcionalidad; estarán contentos de observar un progreso visible hacia su sistema. También pueden contribuir a medida que el proyecto progresa, incrementando su apoyo. Es probable que estos mismos usuarios sean quienes le digan lo cerca que está del objetivo cada iteración.

- **Los desarrolladores construyen una estructura en la que trabajar:** El trozo de papel más abrumador es el que no tiene nada escrito. Si ya ha determinado todas las interacciones de extremo a extremo de su aplicación y las ha representado en el código, entonces su equipo no necesita sacar tanto de la nada. Eso hace que todo el mundo sea más productivo y fomenta la consistencia.
- **Tiene una plataforma de integración:** Como el sistema está conectado de extremo a extremo, tiene un entorno en el que puede añadir nuevas porciones de código una vez que hayan pasado las pruebas unitarias. En vez de intentar una integración de tipo *big bang*, estará integrando cada día (con frecuencia, muchas veces al día). El impacto de cada nuevo cambio es más evidente y las interacciones son más limitadas, así que la depuración y las pruebas son más rápidas y exactas.
- **Tiene algo para mostrar:** Los patrocinadores de los proyectos y los mandamases tienen tendencia a querer ver demos en los momentos más inoportunos. Con el código trazador, siempre tendrá algo que mostrarles.
- **Tiene una mayor sensación de progreso:** En el desarrollo de un código trazador, los desarrolladores abordan los casos de uso uno por uno. Cuando han acabado uno, pasan al siguiente. Es mucho más fácil medir el rendimiento y demostrar el progreso al usuario. Como cada desarrollo individual es más pequeño, evita crear esos bloques monolíticos de código que se anuncian como completados al 95 % semana tras semana.

Las balas trazadoras no siempre dan en el blanco

Las balas trazadoras muestran dónde estamos dando, pero puede que eso no siempre sea la diana. Lo que hacemos entonces es ajustar nuestra mira telescópica hasta que está en el blanco. Esa es la cuestión.

Ocurre lo mismo con el código trazador. Usamos la técnica en situaciones en las que no estamos 100 % seguros de hacia dónde vamos. No debería sorprenderse si falla el primer par de intentos: el usuario dice “no me refería a eso” o los datos que necesita no están disponibles cuando los necesita, o parece probable que haya problemas de rendimiento. Cambie lo que tiene para que se acerque más al objetivo y alégrese de haber utilizado una metodología de desarrollo Lean; una porción pequeña de código tiene una inercia baja; es fácil y rápido de cambiar. Podrá reunir *feedback* sobre su aplicación y generar una versión nueva más precisa de forma rápida y barata. Y, como todos los componentes importantes de la aplicación están representados en el código trazador, sus usuarios pueden estar seguros de que lo que están viendo se basa en la realidad, no solo en una especificación sobre el papel.

Código trazador versus creación de prototipos

Puede que le parezca que este concepto del código trazador no es nada más que la creación de prototipos, pero con un nombre agresivo. Hay una diferencia. Con un prototipo, su objetivo es explorar aspectos específicos del sistema final. Con un auténtico prototipo, se deshará de todo lo que haya juntado cuando probaba el concepto, y volverá a escribir su código de manera adecuada utilizando las lecciones que haya aprendido.

Por ejemplo, supongamos que está creando una aplicación que ayuda a los expedidores a determinar cómo colocar cajas de tamaños inusuales en contenedores. Entre otros problemas, la interfaz de usuario tiene que ser intuitiva y los algoritmos que usa para determinar la colocación óptima son muy complejos.

Podría crear un prototipo de una interfaz de usuario para sus usuarios finales en una herramienta de UI. Podría escribir solo el código suficiente para hacer que la interfaz responda a las acciones del usuario. Una vez que acepten el diseño, puede desechar eso y volver a escribir el código con la lógica de negocio detrás, usando el lenguaje objetivo. De manera similar,

podría querer crear un prototipo de un número de algoritmos que realicen la colocación real. Podría escribir código para pruebas funcionales en un lenguaje permisivo de alto nivel, como Python, y código para pruebas de rendimiento de bajo nivel en algo más cercano a la máquina. En cualquier caso, una vez que hubiese tomado su decisión, empezaría de nuevo y escribiría el código de los algoritmos en su entorno final, interactuando con el mundo real. Esto es crear prototipos, y resulta muy útil.

El código trazador se centra en un problema diferente. Necesita saber cómo funciona la aplicación como conjunto. Quiere mostrar a sus usuarios cómo funcionarán las interacciones en la práctica y quiere dar a sus desarrolladores un esqueleto arquitectónico en el que poner el código. En este caso, podría construir un trazador que consistiese en una implementación trivial del algoritmo de colocación en el contenedor (quizá algo como FCFS) y una interfaz de usuario sencilla pero funcional. Una vez que tenga todos los componentes de la aplicación conectados, tendrá un *framework* que mostrar a sus usuarios y desarrolladores. Con el tiempo, añadirá a este *framework* nueva funcionalidad, completando rutinas interrumpidas. Pero el *framework* permanece intacto y usted sabe que el sistema seguirá comportándose como lo hacía cuando se completó el código trazador por primera vez.

La distinción es lo bastante importante para repetirla. Los prototipos generan código desechable. El código trazador es austero, pero está completo y forma parte del esqueleto del sistema final. Imagine la creación de prototipos como el reconocimiento y la recopilación de información que se lleva a cabo antes de que se dispare una sola bala trazadora.

Las secciones relacionadas incluyen

- Tema 13, “Prototipos y notas en *post-its*”.
- Tema 27, “No vaya más rápido que sus faros”.
- Tema 40, “Refactorización”.
- Tema 49, “Equipos pragmáticos”.
- Tema 50, “Los cocos no sirven”.
- Tema 51, “Kit pragmático básico”.
- Tema 52, “Deleite a sus usuarios”.

13 Prototipos y notas en *post-its*

Muchas industrias utilizan prototipos para probar ideas específicas; crear prototipos es mucho más barato que la producción a escala real. Los fabricantes de coches, por ejemplo, pueden construir muchos prototipos diferentes de un diseño de coche nuevo. Cada uno está diseñado para probar un aspecto específico del coche: la aerodinámica, el estilo, las características estructurales, etc. Los de la vieja escuela pueden usar un modelo de arcilla para la prueba en el túnel de viento, quizá un modelo en madera de balsa y cinta aislante para el departamento de arte, etc. Los menos románticos crearán sus modelos en la pantalla de un ordenador o en realidad virtual, reduciendo todavía más los costes. De este modo, los elementos de riesgo o inciertos pueden probarse sin que haya un compromiso con la construcción del objeto real.

Creamos prototipos de software de la misma manera y por las mismas razones: analizar y exponer los riesgos y ofrecer oportunidades para la corrección a un coste muy reducido. Al igual que los fabricantes de coches, podemos crear un prototipo cuyo objetivo sea probar uno o más aspectos específicos de un proyecto.

Tendemos a pensar en los prototipos como algo basado en el código, pero no siempre tiene por qué ser así. Al igual que los fabricantes de coches, podemos construir prototipos con materiales diferentes. Los *post-its* son geniales para crear prototipos de cosas dinámicas, como el flujo de trabajo y la lógica de la aplicación. El prototipo de una interfaz de usuario puede ser un dibujo en una pizarra, como una réplica no funcional hecha con un programa de dibujo o con un constructor de interfaz.

Los prototipos están diseñados para responder solo a unas pocas preguntas, así que son mucho más baratos y rápidos de desarrollar que las aplicaciones que van a producción. El código puede ignorar detalles que sean poco importantes (poco importantes para usted en el momento, pero, probablemente, muy importantes para el usuario más adelante). Si está creando un prototipo para una interfaz de usuario, por ejemplo, puede irse de rositas con datos o resultados incorrectos. Por otra parte, si está investigando aspectos computacionales o de rendimiento, no pasa nada por

tener una interfaz de usuario muy pobre o incluso por no tener interfaz de usuario.

Pero, si se encuentra en un entorno en el que no puede renunciar a los detalles, entonces debe preguntarse si de verdad está construyendo un prototipo. Quizá un estilo de desarrollo como el de las balas trazadoras sería más apropiado en este caso (consulte el tema 12, “Balas trazadoras”).

Cosas para las que puede crear prototipos

¿Qué tipos de cosas podría elegir para investigar con un prototipo? Cualquier cosa que conlleve riesgos. Cualquier cosa que no se haya probado antes, o que sea absolutamente crucial para el sistema final. Cualquier cosa no probada, experimental o dudosa. Cualquier cosa con la que no esté cómodo. Puede crear prototipos de:

- Arquitectura.
- Funcionalidad nueva en un sistema existente.
- Estructura o contenidos de datos externos.
- Componentes o herramientas de terceros.
- Cuestiones de rendimiento.
- Diseño de la interfaz de usuario.

La creación de prototipos es una experiencia de aprendizaje. Su valor no reside en el código producido, sino en las lecciones aprendidas. Ese es el auténtico sentido de la creación de prototipos.

Truco 21. Cree prototipos para aprender.

Cómo utilizar prototipos

Cuando crea un prototipo, ¿qué detalles puede ignorar?

- **Corrección:** Puede utilizar datos *dummy* cuando sea apropiado.
- **Completitud:** El prototipo puede funcionar solo en un sentido muy limitado, quizá con una sola porción preseleccionada de los datos de

entrada y un elemento del menú.

- **Robustez:** Es probable que la comprobación de errores no se complete o no se haga, sin más. Si se desvía del camino predefinido, puede que el prototipo se estrelle y arda entre gloriosos fuegos artificiales. Está bien.
- **Estilo:** El código prototipo no debería tener muchos comentarios ni documentación (aunque puede producir resmas de documentación como resultado de su experiencia con el prototipo).

Los prototipos tratan por encima los detalles y se centran en aspectos específicos del sistema que está considerándose, así que puede que quiera implementarlos utilizando un lenguaje de *scripts* de alto nivel, más alto que el resto del proyecto (quizá un lenguaje como Python o Ruby), ya que estos lenguajes no le estorbarán. Puede elegir seguir desarrollando en el lenguaje utilizado para el prototipo o cambiar; al fin y al cabo, va a desechar el prototipo de todos modos.

Para crear prototipos de interfaces de usuario, utilice una herramienta que le permita centrarse en el aspecto o las interacciones sin preocuparse del código o el marcado.

Los lenguajes de *scripts* también funcionan bien como el "pegamento" para unir partes de bajo nivel en combinaciones nuevas. Usando este enfoque, puede ensamblar con rapidez componentes existentes y crear configuraciones nuevas para ver cómo funcionan las cosas.

Crear prototipos de arquitectura

Muchos prototipos se construyen para dar forma al sistema completo que está considerándose. Al contrario que las balas trazadoras, ninguno de los módulos individuales del sistema prototipo necesita ser especialmente funcional. De hecho, puede que ni siquiera tenga que escribir código para hacer un prototipo de la arquitectura; puede hacer un prototipo en una pizarra, con *post-its* o con fichas. Lo que quiere saber es cómo se sostiene el sistema como conjunto, posponiendo los detalles de nuevo. Estas son algunas de las áreas específicas que puede que quiera comprobar en el prototipo arquitectónico:

- ¿Están bien definidas y son apropiadas las responsabilidades de las áreas principales?
- ¿Están bien definidas las colaboraciones entre los componentes principales?
- ¿Está el acoplamiento minimizado?
- ¿Puede identificar fuentes potenciales de duplicación?
- ¿Son las definiciones y restricciones de la interfaz aceptables?
- ¿Tienen todos los módulos una ruta de acceso a los datos que necesitan durante la ejecución? ¿Tienen ese acceso cuando lo necesitan?

Esta última cuestión tiende a ser la que genera más sorpresas y resultados más valiosos en la experiencia de la creación de prototipos.

Cómo no utilizar prototipos

Antes de que se embarque en cualquier proceso de creación de prototipos basados en código, asegúrese de que todo el mundo entiende que se va a escribir código desechable. Los prototipos pueden resultar atractivos, de manera engañosa, para las personas que no saben que son solo prototipos. Debe dejar muy claro que ese código es desechable, está incompleto y no puede completarse. Es fácil dejarse engañar por la aparente completitud de un prototipo del que se ha hecho una demostración, y puede que los patrocinadores del proyecto o los directores insistan en desplegar el prototipo (o su progenie) si no establece las expectativas correctas. Recuérdeles que se puede crear un prototipo genial de un coche nuevo con madera de balsa y cinta aislante, ¡pero nadie intentaría conducirlo por la calle en hora punta!

Si siente que hay muchas posibilidades en su entorno o cultura de que el propósito del código prototipo pueda malinterpretarse, quizá le convenga más el enfoque de las balas trazadoras. Acabará consiguiendo un *framework* sólido en el que basar el futuro desarrollo.

Los prototipos bien utilizados pueden ahorrarle grandes cantidades de tiempo, dinero y quebraderos de cabeza al identificar y corregir puntos problemáticos potenciales al principio del ciclo de desarrollo, momento en que reparar errores es barato y fácil.

Las secciones relacionadas incluyen

- Tema 12, “Balas trazadoras”.
- Tema 14, “Lenguajes de dominio”.
- Tema 17, “Jugar con el intérprete de comandos”.
- Tema 27, “No vaya más rápido que sus faros”.
- Tema 37, “Escuche a su cerebro reptiliano”.
- Tema 45, “El pozo de los requisitos”.
- Tema 52, “Deleite a sus usuarios”.

Ejercicios

Ejercicio 3

A los del departamento de marketing les gustaría sentarse y pensar algunos diseños de página web con usted. La idea que tienen es la de unos mapas de imagen en los que se puede hacer clic que te lleven a otras páginas, y así sucesivamente. Pero no se deciden por un modelo para la imagen; podría ser un coche, un teléfono o una casa. Tiene una lista de páginas de destino y contenido; les gustaría ver algunos prototipos. Oh, por cierto, tiene 15 minutos. ¿Qué herramientas podría utilizar?

14 Lenguajes de dominio

> *Los límites de mi lenguaje significan los límites de mi mundo.*
>
> *—Ludwig Wittgenstein.*

Los lenguajes informáticos influyen en el modo en que pensamos en un problema y en cómo pensamos en la comunicación. Cada lenguaje viene con una lista de características: palabras de moda como tipado estático frente a tipado dinámico, vinculación temprana frente a vinculación tardía, funcional frente a OO, modelos de herencia, *mixins*, macros. Todas ellas pueden sugerir o esconder determinadas soluciones. Diseñar una solución con C++ en mente producirá resultados diferentes a una solución basada en un pensamiento de estilo Haskell y viceversa. En cambio, y algo que nos

parece más importante, el lenguaje del dominio del problema también puede sugerir una solución de programación.

Siempre intentamos escribir código utilizando el vocabulario del dominio de la aplicación (consulte “Mantenga un glosario”). En algunos casos, los programadores pragmáticos pueden pasar al siguiente nivel y programar usando de verdad el vocabulario, la sintaxis y la semántica (el lenguaje) del dominio.

Truco 22. Programe cerca del dominio del problema.

Algunos lenguajes de dominio del mundo real

Vamos a ver algunos ejemplos donde la gente ha hecho justo eso.

RSpec

RSpec[7] es una biblioteca de pruebas para Ruby. Ha inspirado versiones para la mayoría de los demás lenguajes modernos. Una prueba en RSpec tiene el propósito de reflejar el comportamiento que esperamos de nuestro código.

```
describe BowlingScore do
  it “totals 12 if you score 3 four times” do
    score = BowlingScore.new
    4.times { score.add_pins(3) }
    expect(score.total).to eq(12)
  end
end
```

Cucumber

Cucumber[8] es un modo neutro de lenguaje de programación para especificar pruebas. Podemos ejecutar las pruebas utilizando una versión de Cucumber apropiada para el lenguaje que estamos utilizando. Para soportar la sintaxis de tipo lenguaje neutral, también tenemos que escribir *matchers* específicos que reconozcan frases y parámetros exactos para las pruebas.

```
Feature: Scoring

Background:
  Given an empty scorecard

Scenario: bowling a lot of 3s
  Given I throw a 3
  And I throw a 3
  And I throw a 3
  And I throw a 3
  Then the score should be 12
```

Las pruebas de Cucumber están pensadas para que las lean los clientes del software (aunque, en la práctica, eso ocurre con bastante poca frecuencia; la siguiente nota es una reflexión acerca de por qué puede que pase eso).

¿Por qué muchos usuarios de negocios no leen las características de Cucumber?

Una de las razones por las que el clásico enfoque de recopilar requisitos, diseñar, escribir código y enviar no funciona es que está anclado por el concepto de que sabemos cuáles son los requisitos. Pero rara vez lo sabemos. Los usuarios del negocio tendrán una idea vaga de lo que quieren conseguir, pero ni conocen los detalles ni les importan. Eso es parte de nuestro valor: intuimos la intención y la convertimos en código. Así pues, cuando obligamos a alguien de la parte empresarial a dar el visto bueno a un documento de requisitos o conseguimos que estén de acuerdo con un conjunto de características de Cucumber, es como si estuviésemos pidiéndoles que corrigiesen la ortografía de un ensayo escrito en sumerio. Harán algunos cambios para quedar bien y le darán su aprobación para que nos vayamos de su despacho. Sin embargo, si les damos código que se ejecuta, pueden jugar con él. Ahí es donde saldrán a la superficie sus verdaderas necesidades.

Rutas de Phoenix

Muchos *frameworks* web tienen una herramienta de enrutamiento que asigna peticiones HTTP entrantes a funciones de gestión en el código. Veamos un ejemplo de Phoenix.[9]

```
scope “/”, HelloPhoenix do
  pipe_through :browser # Usar la pila de navegación predeterminada
```

```
  get “/”, PageController, :index
  resources “/users”, UserController
end
```

Esto indica que las peticiones que empiezan con “/” se ejecutarán a través de una serie de filtros apropiados para navegadores. Una petición a “/” en sí la manejará la función `index` en el módulo `PageController`. `UsersController` implementa las funciones necesarias para gestionar un recurso accesible mediante la URL `/users`.

Ansible

Ansible[10] es una herramienta que configura software, por lo general en un montón de servidores remotos. Para ello, lee una especificación que le proporcionamos y, después, hace lo que sea necesario en los servidores para conseguir que reproduzcan esa especificación. La especificación puede escribirse en YAML,[11] un lenguaje que construye estructuras de datos a partir de descripciones de texto:

```
—-
- name: install nginx
apt: name=nginx state=latest

- name: ensure nginx is running (and enable it at boot)
service: name=nginx state=started enabled=yes

- name: write the nginx config file
template: src=templates/nginx.conf.j2 dest=/etc/nginx/nginx.conf
notify:
–restart nginx
```

Este ejemplo garantiza que la versión más reciente de nginx está instalada en mis servidores, que se inicia por defecto y que utiliza un archivo de configuración que usted ha proporcionado.

Características de los lenguajes de dominio

Vamos a ver estos ejemplos con más detalle.

RSpec y el enrutador Phoenix se escriben en sus lenguajes anfitriones (Ruby y Elixir). Emplean algo de código bastante retorcido, incluyendo

metaprogramación y macros, pero al final se compilan y ejecutan como código normal. Las pruebas de Cucumber y las configuraciones de Ansible se escriben en sus propios lenguajes. Una prueba de Cucumber se convierte en código para ejecutarse o en una estructura de datos, mientras que las especificaciones Ansible siempre se convierten en una estructura de datos que es ejecutada por el propio Ansible. Como resultado, RSpec y el código enrutador se incrustan en el código que ejecutamos: son verdaderas extensiones del vocabulario de nuestro código. Cucumber y Ansible son leídos por códigos y convertidos a alguna forma que el código pueda utilizar.

Llamamos a RSpec y al enrutador ejemplos de lenguaje de dominio interno, mientras que Cucumber y Ansible usan lenguajes externos.

Compensaciones entre lenguajes internos y externos

En general, un lenguaje de dominio interno puede aprovechar las características de su lenguaje anfitrión: el lenguaje de dominio que creamos es más potente, y esa potencia es gratis. Por ejemplo, podríamos utilizar algo de código Ruby para crear varias pruebas RSpec de forma automática. En este caso, podemos probar puntuaciones de una partida de bolos cuando no hay semiplenos ni plenos:

```
describe BowlingScore do
  (0..4).each do |pins|
    (1..20).each do |throws|
      target = pins * throws

      it “totals #{target} if you score #{pins} #{throws} times”
do
        score = BowlingScore.new
        throws.times { score.add_pins(pins) }
        expect(score.total).to eq(target)
      end
    end
  end
end
```

Acabamos de escribir 100 pruebas. Podemos tomarnos el resto del día libre.

El inconveniente de los lenguajes de dominio internos es que estamos limitados por la sintaxis y la semántica de ese lenguaje. Aunque algunos lenguajes son bastante flexibles en este aspecto, seguimos viéndonos obligados a llegar a un equilibrio entre el lenguaje que queremos y el lenguaje que podemos implementar.

Al final, cualquier cosa que se nos ocurra debe tener una sintaxis válida en nuestro lenguaje de destino. Los lenguajes con macros (como Elixir, Clojure y Crystal) nos ofrecen algo más de flexibilidad, pero la sintaxis es la sintaxis.

Los lenguajes externos no tienen esas restricciones. Siempre y cuando podamos escribir un analizador sintáctico, nos irá bien. A veces, podemos usar el analizador sintáctico de otros (como ha hecho Ansible al utilizar YAML), pero después volvemos a tener que llegar a un compromiso. Es probable que escribir un analizador sintáctico suponga añadir nuevas bibliotecas y, posiblemente, herramientas a la aplicación. Y escribir un buen analizador no es un trabajo trivial, pero, si se siente valiente, podría fijarse en generadores de analizadores sintácticos como bison o ANTLR, y *frameworks* de análisis sintáctico como los muchos analizadores PEG que existen. Nuestra sugerencia es bastante simple: no dedique más esfuerzo del que ahorra. Escribir un lenguaje de dominio añade costes al proyecto y debe estar convencido de que hay un ahorro que lo compensa (de forma potencial, a largo plazo).

En general, utilice lenguajes externos estándar (como YAML, JSON o CSV) si puede. Si no, fíjese en los lenguajes internos. Recomendaríamos utilizar lenguajes externos solo en casos en los que su lenguaje va a ser escrito por los usuarios de su aplicación.

Un lenguaje de dominio interno de bajo coste

Por último, existe una pequeña trampa para crear lenguajes de dominio internos si no le importa que la sintaxis del lenguaje anfitrión se filtre. No haga un montón de metaprogramación. En vez de eso, escriba funciones para que hagan el trabajo. De hecho, eso es más o menos lo que hace RSpec:

```
describe BowlingScore do
```

```
  it “totals 12 if you score 3 four times” do
    score = BowlingScore.new
    4.times { score.add_pins(3) }
    expect(score.total).to eq(12)
  end
end
```

En este código, `describe`, `it`, `expect`, `to` y `eq` son solo métodos de Ruby. Hay algo de fontanería entre bastidores en lo que respecta al modo en que se pasan los objetos, pero todo es código. Lo exploraremos un poco en los ejercicios.

Las secciones relacionadas incluyen

- Tema 8, “La esencia del buen diseño”.
- Tema 13, “Prototipos y notas en *post-its*”.
- Tema 32, “Configuración”.

Retos

- ¿Podrían expresarse algunos de los requisitos de su proyecto actual en un lenguaje específico de dominio? ¿Sería posible escribir un compilador o traductor que pudiese generar la mayor parte del código requerido?
- Si decide adoptar minilenguajes como una forma de programar más cercana al dominio del problema, está aceptando que hará falta cierto esfuerzo para implantarlos. ¿Se le ocurre algún modo de poder reutilizar el *framework* que desarrolla para un proyecto en otros?

Ejercicios

Ejercicio 4

Queremos implementar un minilenguaje para controlar un sistema de gráficos tortuga simple. El lenguaje consta de comandos de una sola letra, algunos seguidos de un solo número. Por ejemplo, la siguiente entrada dibujaría un rectángulo:

```
P 2 # selecciona el bolígrafo 2
D # baja el bolígrafo
W 2 # dibuja hacia el oeste 2cm
N 1 # después hacia el norte 1
E 2 # después hacia el este 2
S 1 # después otra vez hacia el sur
U # levanta el bolígrafo
```

Implemente el código que analiza sintácticamente este lenguaje. Debería diseñarse de tal manera que sea fácil añadir comandos nuevos.

Ejercicio 5

En el ejercicio anterior, hemos implementado un analizador sintáctico para el lenguaje de dibujo; se trataba de un leguaje de dominio externo. Ahora, impleméntelo otra vez como un lenguaje interno. No haga nada ingenioso: solo escriba una función para cada uno de los comandos. Puede que tenga que cambiar los nombres de los comandos a minúsculas y tal vez envolverlos dentro de algo para proporcionar algo de contexto.

Ejercicio 6

Diseñe una gramática BNF para analizar sintácticamente una especificación de tiempo. Todos los ejemplos siguientes deberían aceptarse:

```
4pm, 7:38pm, 23:42, 3:16, 3:16am
```

Ejercicio 7

Implemente un analizador sintáctico para la gramática BNF del ejercicio anterior con un generador de analizadores PEG en el lenguaje que prefiera. El resultado debería ser un entero que contenga el número de minutos que pasan de la medianoche.

Ejercicio 8

Implemente el analizador de tiempo con un lenguaje de *scripts* y expresiones regulares.

15 Estimaciones

La Biblioteca del Congreso en Washington, DC, tiene en la actualidad unos 75 terabytes de información digital en línea. ¡Rápido! ¿Cuánto se tardará en enviar toda esa información a través de una red de 1 Gbps? ¿Cuánto almacenamiento necesitará para millones de nombres y direcciones? ¿Cuánto se tarda en comprimir 100 Mb de texto? ¿Cuántos meses tardará en entregar el proyecto?

En cierto nivel, ninguna de estas preguntas tiene sentido; a todas les falta información. Aun así, todas pueden responderse, siempre y cuando esté cómodo haciendo estimaciones. Y, en el proceso de producir una estimación, llegará a entender más acerca del mundo en el que habitan sus programas.

Al aprender a estimar y al desarrollar esta habilidad hasta el punto en que tenga una sensación intuitiva de la magnitud de las cosas, será capaz de mostrar una capacidad que parecerá mágica para determinar su factibilidad. Cuando alguien dice: “Enviaremos la copia de seguridad por una conexión de red a S3”, podrá saber de manera intuitiva si eso es práctico. Cuando esté escribiendo código, será capaz de saber qué subsistemas necesitan optimizarse y cuáles pueden dejarse sin tocar.

Truco 23. Estime para evitar sorpresas.

Como extra, al final de esta sección revelaremos la única respuesta correcta que hay que dar cuando alguien le pide que haga una estimación.

¿Cuánta exactitud es suficiente exactitud?

Hasta cierto punto, todas las respuestas son estimaciones. Simplemente, algunas son más exactas que otras. Así pues, la primera pregunta que debe plantearse cuando alguien le pida que haga una estimación es el contexto en el que se recibirá su respuesta. ¿Necesitan mucha exactitud o les interesa una cifra aproximada?

Uno de los aspectos interesantes de las estimaciones es que las unidades que utilicemos suponen una diferencia en la interpretación del resultado. Si decimos que algo llevará unos 130 días laborables, la gente esperará resultados bastante pronto. Sin embargo, si decimos que llevará unos seis

meses, entonces saben que tienen que esperar entre cinco y siete meses a partir de ahora. Ambos números representan la misma duración, pero es probable que “130 días” implique un grado de exactitud más alto de lo que sentimos. Le recomendamos que utilice la siguiente escala para las estimaciones del tiempo:

Duración	Haga estimación en
1-15 días	Días
3-6 semanas	Semanas
8-20 semanas	Meses
20+ semanas	Piénselo mucho antes de ofrecer una estimación

Así pues, si después de hacer todo el trabajo necesario, decide que un proyecto llevará 125 días laborables (25 semanas), puede que le convenga ofrecer una estimación de “unos seis meses”.

Los mismos conceptos se aplican a las estimaciones de cualquier cantidad: elija las unidades de su respuesta de forma que reflejen la exactitud que desea transmitir.

¿De dónde vienen las estimaciones?

Todas las estimaciones se basan en modelos del problema. Pero, antes de que nos adentremos demasiado en las técnicas de creación de modelos, debemos mencionar un truco básico para realizar las estimaciones que siempre ofrece buenas respuestas: pregunte a alguien que ya lo haya hecho. Antes de centrarse demasiado en la creación de un modelo, busque a alguien que haya estado en una situación similar en el pasado. Fíjese en cómo resolvieron su problema.

Es poco probable que encuentre un caso exactamente igual, pero se sorprendería de cuántas veces se puede aprender con éxito de las experiencias de otras personas.

Entienda qué se le está preguntando

La primera parte de cualquier ejercicio de estimación es desarrollar la comprensión de lo que se le está preguntando. Además de las cuestiones de exactitud que hemos mencionado antes, necesita entender el alcance del dominio. A menudo, esto está implícito en la pregunta, pero es necesario que convierta en una costumbre pensar en el alcance antes de empezar a hacer suposiciones. Con frecuencia, el alcance que elija formará parte de la respuesta que dé: “Suponiendo que no haya accidentes de tráfico y que haya combustible en el coche, debería estar ahí en 20 minutos”.

Cree un modelo del sistema

Esta es la parte divertida de la estimación. A partir de su comprensión de la pregunta que se está planteando, cree un modelo mental básico. Si está estimando tiempos de respuesta, el modelo puede incluir un servidor y algún tipo de tráfico de llegada. Para un proyecto, el modelo puede ser los pasos que su organización utiliza durante el desarrollo, junto con una imagen muy básica de cómo podría implementarse el sistema.

La creación del modelo puede ser tanto creativa como útil a largo plazo. A menudo, el proceso de creación del modelo conduce al descubrimiento de patrones y procesos subyacentes que no eran visibles en la superficie. Puede incluso que quiera replantearse la pregunta original: “Me pidieron una estimación para hacer X. Sin embargo, parece ser que Y, una variante de X, podría hacerse en la mitad de tiempo y solo perderían una característica”.

Crear el modelo introduce inexactitudes en el proceso de estimación. Es algo inevitable, pero también beneficioso. Está sacrificando la simplicidad del modelo en favor de la exactitud. Puede que duplicar el esfuerzo en el modelo le proporcione solo un ligero incremento de la exactitud. Su experiencia le dirá cuándo dejar de pulirlo.

Descomponga el modelo en componentes

Una vez que tiene un modelo, puede descomponerlo en componentes. Necesitará descubrir las reglas matemáticas que describen la manera en que interactúan estos componentes. A veces un componente aporta un único valor que se añade al resultado. Algunos componentes pueden proporcionar factores multiplicadores, mientras que otros pueden ser más complicados

(como aquellos que simulan la llegada de tráfico a un nodo). Verá que cada componente suele tener parámetros que afectan a cómo contribuye al modelo global. En esta etapa, simplemente identifique cada parámetro.

Dé a cada parámetro un valor

Una vez que hemos determinado los parámetros, podemos examinarlos y asignar a cada uno un valor. Espere introducir algunos errores en este paso. El truco es averiguar qué parámetros tienen más impacto sobre el resultado y concentrarse en acertar con ellos. Por lo general, los parámetros cuyos valores se suman para dar un resultado son menos significativos que aquellos que se multiplican o se dividen. Duplicar la velocidad de una línea puede duplicar la cantidad de datos recibidos en una hora, mientras que sumar un retardo de tránsito de 5 ms no tendrá ningún efecto notable. Debería tener una manera justificable de calcular estos parámetros críticos. Para el ejemplo de la cola, podría convenirle medir la tasa de llegada de la transacción real del sistema existente o encontrar un sistema parecido para medir. De manera similar, podría medir el tiempo actual que se tarda en atender una solicitud o elaborar una estimación usando las técnicas descritas en esta sección. De hecho, a menudo verá que basa sus estimaciones en otras subestimaciones. Aquí es donde cometerá los mayores errores.

Calcule las respuestas

Solo en los casos más simples la estimación tendrá una única respuesta. Puede estar satisfecho con decir: “Puedo recorrer cinco manzanas a pie en 15 minutos”. Sin embargo, a medida que los sistemas se vuelven más complejos, tendrá que dar más vueltas a sus respuestas. Realice múltiples cálculos, variando los valores de los parámetros críticos, hasta que determine cuáles son los que realmente guían el modelo. Una hoja de cálculo puede resultar de gran ayuda. Después, formule su respuesta con respecto a estos parámetros. “El tiempo de respuesta es más o menos tres cuartos de segundo si el sistema tiene SSD y 32 GB de memoria y un segundo con 16 GB de memoria”. (Fíjese en que “tres cuartos de segundo” transmite una sensación de exactitud diferente a 750 ms).

Durante la fase de los cálculos, recibirá respuestas que parecerán extrañas. No tenga prisa por desecharlas. Si su aritmética es correcta, es probable que su comprensión del problema o su modelo estén mal. Esa información es valiosa.

Haga un seguimiento de su pericia para hacer estimaciones

Creemos que es buena idea registrar sus estimaciones para que pueda ver cuánto se ha acercado. Si una estimación general implicaba calcular subestimaciones, haga también un seguimiento de ellas. A menudo, descubrirá que sus estimaciones eran bastante buenas; de hecho, después de un tiempo, llegará a esperarlo.

Cuando una estimación resulte ser errónea, no se encoja de hombros y a otra cosa, mariposa; averigüe el motivo. Quizá eligió algunos parámetros que no se correspondían con la realidad del problema. Tal vez su modelo estaba mal. Fuese cual fuese la razón, tómese un tiempo para descubrir lo que ha pasado. Si lo hace, su siguiente estimación será mejor.

Estimar el calendario de los proyectos

Por lo general, se le pedirá que estime cuánto tiempo le llevará algo. Si ese "algo" es complejo, la estimación puede ser muy difícil de producir. En esta sección, veremos dos técnicas para reducir esa incertidumbre.

Pintar el misil

> *—¿Cuánto tiempo se tardará en pintar la casa?*
> *—Bueno, si todo va bien y esta pintura tiene la cobertura que anuncia, podría llevar solo 10 horas. Pero eso es poco probable: creo que una cifra más realista se acerca más a las 18 horas. Y, por supuesto, si hace mal tiempo, se podría llegar a las 30 horas o más.*

Así es como la gente hace estimaciones en el mundo real. No con un solo número (a menos que se le obligue a decir uno), sino en relación con distintos escenarios.

Cuando la Armada de EE. UU. necesitaba planear el proyecto del submarino Polaris, adoptó este estilo de estimación con una metodología

que denominaban PERT (*Program Evaluation Review Technique*, técnica de revisión y evaluación de programas).

Toda tarea de PERT tiene una estimación optimista, una que se considera la más probable y una pesimista. Las tareas se organizan en una red de dependencia y, después, se une alguna estadística simple para identificar cuáles tienen probabilidades de ser el mejor y el peor tiempo para el proyecto general.

Usar un rango de valores como este es una manera estupenda de evitar una de las causas más comunes de los errores de estimación: rellenar un número porque no estamos seguros. En vez de eso, las estadísticas que hay detrás de PERT distribuyen la incertidumbre por usted y le proporcionan mejores estimaciones de todo el proyecto.

Sin embargo, a nosotros no nos entusiasma mucho esto. La gente tiende a producir gráficos que ocupan paredes enteras con todas las tareas del proyecto y creen de forma implícita que, solo por haber usado una fórmula, tienen una estimación exacta. Lo más probable es que no sea así, porque nunca han hecho eso antes.

Comerse el elefante

Consideramos que, a menudo, la única manera de determinar el calendario para un proyecto es adquirir experiencia en ese mismo proyecto. Eso no tiene por qué ser una paradoja si practica un desarrollo gradual, repitiendo los mismos pasos con porciones muy pequeñas de funcionalidad:

- Comprobar los requisitos.
- Analizar el riesgo (y priorizar antes los elementos de mayor riesgo).
- Diseñar, implementar, integrar.
- Validar con los usuarios.

Al principio, puede que solo tenga una vaga idea de cuántas interacciones se requerirán o de lo largas que serán. Algunos métodos requieren que determine esto como parte del plan inicial; sin embargo, eso es un error para todos los proyectos, salvo los más triviales. A menos que esté haciendo una aplicación similar a una anterior, con el mismo equipo y con la misma tecnología, solo estará haciendo conjeturas.

Así pues, complete la creación del código y las pruebas y marque eso como el final de la primera iteración. Basándose en esa experiencia, puede perfeccionar la suposición inicial acerca del número de iteraciones y lo que se puede incluir en cada una. Ese perfeccionamiento va mejorando cada vez, y la confianza en el calendario crece con él. Este tipo de estimación se realiza con frecuencia durante la revisión del equipo al final de cada ciclo iterativo.

También es así como, según el viejo chiste, se come un elefante: bocado a bocado.

Truco 24. Itere el calendario con el código.

Puede que esto no guste mucho a los directores, que suelen querer un único número estricto antes incluso de que empiece el proyecto. Tendrá que ayudarles a entender que el equipo, su productividad y el entorno determinarán el calendario. Al formalizar esto y perfeccionar el calendario como parte de cada iteración, les estará ofreciendo las estimaciones más exactas que puede sobre el calendario.

Qué decir cuando le pidan una estimación

Tiene que decir: “Luego le respondo”.

Casi siempre obtendrá mejores resultados si ralentiza el proceso y dedica un tiempo a reflexionar sobre los pasos que describimos en esta sección. Las estimaciones que se ofrecen al lado de la máquina de café volverán para atormentarle (como el café).

Las secciones relacionadas incluyen

- Tema 7, “¡Comuníquese!”.
- Tema 39, “Velocidad de los algoritmos”.

Retos

- Empiece a llevar un registro de sus estimaciones. Para cada una, anote lo exacta que resultó ser. Si se equivocó por más de un 50 %, intente averiguar dónde falló la estimación.

Ejercicios

Ejercicio 9

Se le pregunta: “¿Qué tiene mayor ancho de banda: una conexión de red de 1 Gbps o una persona que camina entre dos ordenadores con un dispositivo de almacenamiento de 1 TB lleno en su bolsillo?”. ¿Qué limitaciones pondrá a su contestación para garantizar que el alcance de su respuesta es correcto? (Por ejemplo, podría decir que el tiempo invertido en acceder al dispositivo de almacenamiento se ignora).

Ejercicio 10

Entonces, ¿cuál tiene mayor ancho de banda?

---

[1] Parafraseando una antigua canción de Arlen/Mercer...

[2] O, a lo mejor, para mantener su cordura, después de cada 10 veces...

[3] https://github.com/OAI/OpenAPI-Specification.

[4] En realidad, esto es algo ingenuo. A menos que tenga una suerte fuera de lo común, la mayoría de los cambios de requisitos en el mundo real afectarán a múltiples funciones en el sistema. Sin embargo, si analiza el cambio en lo que respecta a las funciones, cada cambio funcional debería afectar solo, de manera ideal, a un módulo.

[5] De hecho, este libro está escrito en Markdown, y compuesto directamente desde la fuente de Markdown.

[6] Coja un sistema caótico, o no lineal, y aplique un cambio pequeño en una de sus entradas. Puede que obtenga un resultado grande y, a menudo, impredecible. La típica mariposa que bate las alas en Tokio podría poner en marcha una cadena de acontecimientos que acabe generando un tornado en Texas. ¿Le suena a algún proyecto que conozca?

[7] https://rspec.info.

[8] https://cucumber.io/.

[9] https://phoenixframework.org/.

[10] https://www.ansible.com/.

[11] https://yaml.org/.

3
Las herramientas básicas

Todo creador comienza su viaje con un conjunto básico de herramientas de buena calidad. Un carpintero puede necesitar reglas, calibres, un par de sierras, unas buenas garlopas, escoplos finos, taladros, mazos y tornillos de banco. Estas herramientas se elegirán con mimo, estarán hechas paras durar, realizarán trabajos específicos sin mucho solapamiento con otras herramientas y, quizá lo más importante, se sentirá que las manos del carpintero en ciernes son el lugar al que pertenecen.

Comienza entonces el proceso de aprendizaje y adaptación. Cada herramienta tiene su propia personalidad y sus peculiaridades, y necesita su propio manejo especial. Cada una debe afilarse de una manera única o sostenerse de un modo concreto. Con el tiempo, cada una se desgastará en función de su uso, hasta que la empuñadura parezca un molde de las manos del carpintero y la superficie de corte se alinee a la perfección con el ángulo en el que sostiene la herramienta. En este punto, las herramientas se convierten en canales desde el cerebro del carpintero hasta el producto acabado; se han convertido en extensiones de sus manos. Con el tiempo, el carpintero añadirá herramientas nuevas, como engalletadoras, sierras de inglete guiadas por láser, plantillas de cola de pato... todas ellas formas maravillosas de tecnología. Pero puede estar seguro de que el carpintero será más feliz con esas herramientas originales en la mano, sintiendo cómo canta la garlopa al deslizarse por la madera.

Las herramientas amplifican el talento. Cuanto mejores sean sus herramientas y mejor sepa cómo usarlas, más productivo podrá ser. Empiece con un conjunto básico de herramientas aplicables a nivel general. A medida que adquiera experiencia y vaya encontrándose requisitos especiales, añada más al conjunto básico. Al igual que el carpintero, cuente con añadir herramientas a su kit con regularidad. Esté siempre atento a la aparición de formas mejores de hacer las cosas. Si se encuentra en una situación en la que siente que sus herramientas actuales no sirven, tome

nota de buscar algo diferente o más potente que le hubiese ayudado. Deje que la necesidad impulse sus adquisiciones.

Muchos programadores nuevos cometen el error de adoptar una sola herramienta potente, como un entorno de desarrollo integrado (IDE, por sus siglas en inglés) y nunca abandonan su acogedora interfaz. Eso es un verdadero error. Tiene que sentirse cómodo más allá de los límites impuestos por un IDE. La única manera de hacerlo es mantener las herramientas básicas afiladas y listas para usar.

En este capítulo, hablaremos acerca de la investigación de su propia caja de herramientas básica. Como ocurre con cualquier buena charla sobre herramientas, empezaremos (en “El poder del texto simple”) echando un vistazo a nuestra materia prima, aquello a lo que vamos a dar forma. Desde ahí, pasaremos a la mesa de trabajo o, en nuestro caso, el ordenador. ¿Cómo puede utilizar su ordenador para aprovechar al máximo las herramientas que utiliza? Lo veremos en “Jugar con el intérprete de comandos”. Ahora que tenemos el material y una mesa en la que trabajar, vamos a fijarnos en la herramienta que probablemente vaya a utilizar más que cualquier otra, su editor. En “Edición potente”, sugeriremos la manera de hacer que sea más eficiente.

Para garantizar que nunca se pierde nada de nuestro valioso trabajo, deberíamos utilizar un sistema de “Control de versiones”, incluso para cosas personales como recetas o notas. Y, puesto que Murphy era en realidad un optimista, al fin y al cabo, no se puede ser un gran programador a menos que desarrolle una gran habilidad en la “Depuración”.

Necesitará algo de pegamento para unir toda la magia. Veremos algunas posibilidades en “Manipulación de texto”.

Por último, vale más tinta pálida que memoria brillante. Mantenga un registro de sus pensamientos y su historial, como describimos en “Cuadernos de bitácora de ingeniería”.

Dedique tiempo a aprender a utilizar estas herramientas y, en algún momento, se sorprenderá al descubrir sus dedos moviéndose sobre el teclado, manipulando texto sin un pensamiento consciente. Las herramientas se habrán convertido en una extensión de sus manos.

16 El poder del texto simple

Como programadores pragmáticos, nuestro material básico no es madera o hierro, es conocimiento. Reunimos requisitos como conocimiento y, después, expresamos ese conocimiento en nuestros diseños, implementaciones, pruebas y documentos. Y creemos que el mejor formato para almacenar el conocimiento de manera persistente es el texto simple. Con texto simple, nos damos a nosotros mismos la capacidad para manipular el conocimiento, tanto de forma manual como programática, usando prácticamente cualquier herramienta a nuestra disposición.

El problema con la mayoría de formatos binarios es que el contexto necesario para entender los datos está separado de los datos en sí. Estamos apartando de manera artificial los datos de su significado. Los datos podrían estar también encriptados; no significan absolutamente nada sin la lógica de la aplicación para analizarlos sintácticamente. Sin embargo, con el texto simple, podemos conseguir un flujo de datos autodescriptivo que sea independiente de la aplicación que lo ha creado.

¿Qué es el texto simple?

El texto simple está formado por caracteres imprimibles de una forma que transmite información. Puede ser tan simple como una lista de la compra:

```
* leche
* lechuga
* café
```

o tan complejo como la fuente de este libro (sí, está en texto simple, para disgusto de la editorial, que quería que usásemos un procesador de texto).

La parte de la información es importante. Lo siguiente no es texto simple útil:

```
hlj;uijn bfjxrrctvh jkni’pio6p7gu;vh bjxrdi5rgvhj
```

Y esto tampoco:

```
Field19=467abe
```

El lector no tiene ni idea de cuál podría ser el significado de `467abe`. Nos gusta que nuestro texto simple sea comprensible para los humanos.

Truco 25. Mantenga el conocimiento en texto simple.

El poder del texto

El texto simple no significa que el texto sea desestructurado; HTML, JSON, YAML, etc., son texto simple, al igual que la mayoría de los protocolos fundamentales de la red, HTTP, SMTP, IMAP, etc. Y hay buenas razones para ello:

- Seguro frente a la obsolescencia.
- Aprovechamiento de herramientas existentes.
- Realización de pruebas más fácil.

Seguro frente a la obsolescencia

Las formas de datos legibles para los humanos y los datos autodescriptivos sobrevivirán a todas las demás formas de datos y las aplicaciones que las crearon. Punto. Mientras los datos sobrevivan, tendrá la oportunidad de usarlos; posiblemente, mucho tiempo después de que la aplicación original con la que se escribieron haya desaparecido. Puede analizar sintácticamente un archivo así con solo un conocimiento parcial de su formato; con la mayoría de archivos binarios, debe conocer todos los detalles del formato entero para poder analizarlo sintácticamente con éxito.

Piense en un archivo de datos de algún sistema heredado que se le da.[1] Sabe muy poco de la aplicación original; lo único que es importante para usted es que mantenía una lista de números de la Seguridad Social de los clientes, que tiene que encontrar y extraer. Entre los datos, ve:

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

Al reconocer el formato de un número de la Seguridad Social, puede escribir enseguida un programa pequeño para extraer esos datos, incluso si no tiene información sobre nada más en el archivo.

Pero imagine que el archivo hubiese tenido este formato en vez del anterior:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

Puede que no hubiese reconocido el significado de los números con tanta facilidad. Esa es la diferencia entre datos legibles para los humanos y datos comprensibles para los humanos. Y ya que estamos, `FIELD10` tampoco ayuda mucho. Algo como:

```
<SOCIAL-SECURITY-NO>123-45-6789</SOCIAL-SECURITY-NO>
```

hacen que el ejercicio sea obvio y garantiza que los datos sobrevivirán a cualquier proyecto que los crease.

Aprovechamiento

Casi todas las herramientas del universo de la informática, desde los sistemas de control de versiones a los editores y las herramientas de línea de comandos, pueden manejar texto simple.

Por ejemplo, supongamos que tiene un despliegue a producción de una aplicación grande con un archivo de configuración complejo específico para un sitio. Si el archivo está en texto simple, podría colocarlo bajo un sistema de control de versiones (consulte el tema 19 “Control de versiones”), de manera que mantenga un historial de todos los cambios de forma automática. Las herramientas de comparación de archivos como `diff` y `fc` le permiten descubrir de un solo vistazo qué cambios se han realizado, mientras que `sum` le permite generar un *checksum* para monitorizar el archivo en busca de modificaciones accidentales (o maliciosas).

La filosofía de Unix

Unix es famoso por estar diseñado en torno a la filosofía de herramientas pequeñas y afiladas, cada una de ellas pensada para hacer una cosa bien. Esta filosofía se pone en marcha mediante el uso de un formato subyacente común, el

archivo de texto simple orientado a líneas. Las bases de datos usadas para la administración de sistemas (usuarios y contraseñas, configuración de red, etc.) se mantienen en archivos de texto simple. (Algunos sistemas también mantienen una forma binaria de determinadas bases de datos como optimización del rendimiento. La versión en texto simple se mantiene como una interfaz de la versión binaria).
Cuando un sistema falla, puede que nos encontremos solo con un entorno mínimo para restaurarlo (puede que no podamos acceder a los controladores de gráficos, por ejemplo). Situaciones como esta pueden hacer que aprecie más la sencillez del texto simple. Además, es más fácil realizar búsquedas en archivos de texto simple. Si no recuerda qué archivo de configuración gestiona las copias de seguridad de su sistema, el uso rápido de `grep -r backup /etc` debería indicárselo.

Realización de pruebas más fácil

Si utiliza texto simple para crear datos sintéticos para controlar las pruebas del sistema, solo es cuestión de añadir, actualizar o modificar los datos de prueba sin tener que crear ninguna herramienta especial para hacerlo. De manera similar, la salida en texto simple de pruebas de regresión puede analizarse de manera trivial con comandos del *shell* o un simple *script*.

Mínimo común denominador

Incluso en el futuro de agentes inteligentes basados en *blockchain* que recorren la salvaje y peligrosa Internet de forma autónoma, negociando el intercambio de datos entre ellos, el omnipresente archivo de texto seguirá estando ahí. De hecho, en entornos heterogéneos, las ventajas del texto simple pueden pesar más que los inconvenientes. Tiene que garantizar que todas las partes pueden comunicarse utilizando un estándar común. El texto simple es ese estándar.

Las secciones relacionadas incluyen

- Tema 17, “Jugar con el intérprete de comandos”.
- Tema 21, “Manipulación de texto”.
- Tema 32, “Configuración”.

Retos

- Diseñe una pequeña base de datos para una agenda (nombre, teléfono, etc.) utilizando una representación binaria clara en el lenguaje que quiera. Hágalo antes de leer el resto de este reto.
  - Traduzca ese formato a un formato de texto simple usando XML o JSON.
  - Para cada versión, añada un campo nuevo de longitud variable llamado “indicaciones”, en el que podría introducir instrucciones para llegar a la casa de cada persona.

    ¿Qué problemas surgen respecto al versionado y la extensibilidad? ¿Qué formato es más fácil de modificar? ¿Qué hay de la conversión de los datos existentes?

17 Jugar con el intérprete de comandos

Todo carpintero necesita una buena mesa de trabajo, sólida y fiable, para sujetar las piezas en las que trabaja a una altura conveniente mientras les da forma. La mesa de trabajo se convierte en el centro del taller, y el carpintero vuelve a ella una y otra vez mientras la pieza toma forma.

Para un programador que manipula archivos de texto, esa mesa de trabajo es el intérprete de comandos. Desde el *prompt* del indicador de comandos, puede invocar su repertorio completo de herramientas, usando *pipes* para combinarlos de maneras que sus desarrolladores originales jamás soñaron. Desde el intérprete de comandos, puede iniciar aplicaciones, depuradores, navegadores, editores y servicios. Puede buscar archivos, consultar el estado del sistema y filtrar salidas. Y, al programar el intérprete de comandos, puede crear comandos en macros complejas para actividades que realiza a menudo.

Para los programadores que han crecido con interfaces GUI y entornos de desarrollo integrados, esto podría parecer una posición extrema. Al fin y al cabo, ¿no se puede hacer todo igual de bien apuntando y haciendo clic?

La respuesta sencilla es “no”. Las interfaces GUI son maravillosas y pueden ser más rápidas y convenientes para algunas operaciones sencillas. Mover archivos, leer y escribir correos electrónicos y crear y desarrollar un

proyecto son cosas que podría interesarle hacer en un entorno gráfico, pero, si hace todo su trabajo usando GUI, está perdiéndose todas las capacidades de su entorno. No podrá automatizar tareas comunes ni aprovechar el máximo potencial de las herramientas a su disposición. Y no podrá combinar sus herramientas para crear macros personalizadas. Un beneficio de las GUI es WYSIWYG (*what you see is what you get*, lo que ves es lo que obtienes). La desventaja es WYSIAYG (*what you see is all you get*, lo que ves es todo lo que obtienes).

Los entornos GUI suelen estar limitados a las capacidades que sus diseñadores planearon. Si necesita ir más allá del modelo proporcionado por el diseñador, por lo general no tendrá mucha suerte, y la mayoría de las veces sí necesita ir más allá del modelo. Los programadores pragmáticos no solo escribimos código, desarrollamos modelos de objetos, escribimos documentación o automatizamos el proceso de construcción; hacemos todas esas cosas. El alcance de una herramienta cualquiera suele verse limitado a las tareas que se espera que realice esa herramienta. Por ejemplo, supongamos que necesita integrar un preprocesador de código (para implementar diseño por contrato o pragmas de multiprocesamiento, o algo de ese estilo) en su IDE. A menos que el diseñador del IDE proporcionase de forma explícita enganches para esa capacidad, no podrá hacerlo.

Truco 26. Use el poder de los intérpretes de comandos.

Familiarícese con el intérprete de comandos y verá cómo se dispara su productividad. ¿Necesita crear una lista de todos los nombres de paquetes únicos importados de forma explícita por su código Java? Esto la almacena en un archivo llamado “`list`”:

sh/packages.sh

```
grep '^import ' *.java |
  sed -e's/^import *//' -e's/;.*$//' |
  sort -u >list
```

Si no ha dedicado mucho tiempo a explorar las capacidades del intérprete de comandos en los sistemas que utiliza, esto podría parecer abrumador. No obstante, invierta algo de tiempo en familiarizarse con su intérprete de comandos y pronto todo empezará a tener sentido. Juguetee

con el intérprete de comandos y le sorprenderá cuánto le ayuda a ser más productivo.

Su propio intérprete de comandos

Del mismo modo que un carpintero personalizará su espacio de trabajo, un desarrollador debería personalizar su intérprete de comandos. Esto suele implicar cambiar la configuración del programa de terminal que utilice. Los cambios comunes incluyen:

- **Configurar temas de color:** Puede pasar muchas muchas horas probando todos y cada uno de los temas disponibles en línea para su intérprete de comandos en particular.
- **Configurar un** *prompt***:** El *prompt* que le indica que el intérprete de comandos está listo para que escriba un comando puede configurarse para mostrar casi cualquier información que pueda querer (y un montón de cosas que nunca querría). Aquí las preferencias personales lo son todo: nosotros tendemos a preferir *prompts* simples, con un nombre del directorio actual acortado y el estado del control de versiones junto con la hora.
- **Alias y funciones del intérprete de comandos:** Simplifique su flujo de trabajo convirtiendo comandos que utilice mucho en alias sencillos. Quizá actualiza con regularidad su caja de Linux, pero nunca recuerda si actualiza (*update*) y mejora (*upgrade*) o si mejora y actualiza. Cree un alias:

```
alias apt-up='sudo apt-get update && sudo apt-get upgrade'
```

  También puede ser que haya eliminado archivos por accidente con el comando `rm` con demasiada frecuencia. Escriba un alias que siempre lo pida en el futuro:

```
alias rm ='rm -iv'
```

- **Completitud de comandos:** La mayoría de los intérpretes completarán los nombres de los comandos y archivos: escriba los primeros caracteres, pulse la tecla de tabulación y rellenará lo que pueda. Pero puede llevar esto mucho más lejos, configurando el

intérprete para que reconozca el comando que está introduciendo y ofrezca completitudes específicas de cada contexto. Algunos incluso personalizan la completitud dependiendo del directorio actual.

Pasará mucho tiempo en uno de esos intérpretes de comandos, así que conviértalo en un hogar.

Las secciones relacionadas incluyen

- Tema 13, “Prototipos y notas en *post-its*”.
- Tema 16, “El poder del texto simple”.
- Tema 21, “Manipulación de texto”.
- Tema 30, “Transformar la programación”.
- Tema 51, “Kit pragmático básico”.

Retos

- ¿Hay actualmente cosas que esté haciendo de forma manual en una GUI? ¿Alguna vez pasa instrucciones a colegas que implican varios pasos “haz clic en este botón”, “selecciona este elemento” individuales? ¿Podrían automatizarse?
- Cada vez que pase a un entorno nuevo, haga un esfuerzo para averiguar qué intérpretes de comandos están disponibles. Mire si puede llevarse su intérprete actual con usted.
- Investigue alternativas a su intérprete de comandos actual. Si se topa con un problema que su intérprete no puede abordar, compruebe si un intérprete alternativo podría hacerlo mejor.

18 Edición potente

Antes hemos hablado de que las herramientas son extensiones de nuestras manos. Bueno, esto se aplica a los editores más que a cualquier otra herramienta de software. Tiene que ser capaz de manipular texto con el

menor esfuerzo posible, porque el texto es la materia prima básica de la programación.

En la primera edición de este libro, recomendábamos utilizar un solo editor para todo: código, documentación, memorandos, administración de sistemas, etc. Ahora hemos relajado un poco esta postura. Nos parece bien que utilice tantos editores como quiera. Simplemente, nos gustaría que trabajase por conseguir la fluidez en cada uno de ellos.

Truco 27. Consiga la fluidez en el uso de los editores.

¿Por qué es esto importante? ¿Estamos diciendo que eso le ahorrará mucho tiempo? En realidad, sí: a lo largo de un año, puede que llegue a ganar una semana adicional si hace que su edición sea solo un 4 % más eficiente y edita 20 horas a la semana.

Pero ese no es el beneficio real. No, la ganancia principal es que, al lograr la fluidez, ya no tiene que pensar en la mecánica de la edición. La distancia entre pensar algo y hacer que aparezca en un búfer de edición se reduce. Sus pensamientos fluirán y su programación se beneficiará. (Si alguna vez ha enseñado a alguien a conducir, entenderá la diferencia entre alguien que tiene que pensar en cada acción que realiza y un conductor más experimentado que controla el coche de forma instintiva).

¿Qué significa “fluidez”?

¿Qué cuenta como “fluidez”? Aquí hay una lista de retos:

- Cuando edite texto, mueva y seleccione por carácter, palabra, línea y párrafo.
- Cuando edite código, mueva por varias unidades sintácticas (delimitadores coincidentes, funciones, módulos...).
- Vuelva a sangrar el código tras los cambios.
- Comente y descomente bloques de código con un solo comando.
- Deshaga y rehaga cambios.
- Divida la ventana del editor en múltiples paneles y muévase de unos a otros.

- Vaya a un número de línea en particular.
- Ordene líneas seleccionadas.
- Busque tanto cadenas como expresiones regulares y repita búsquedas anteriores.
- Cree de manera temporal múltiples cursores basados en una selección o en una búsqueda de patrones y edite el texto de cada uno en paralelo.
- Muestre errores de compilación en el proyecto actual.
- Ejecute las pruebas de proyecto actual.

¿Puede hacer todo esto sin usar un ratón/panel táctil?

Podría decir que su editor actual no puede hacer algunas de esas cosas. ¿Quizá ha llegado el momento de cambiar de editor?

Avanzar hacia la fluidez

No creemos que haya más de un puñado de personas que conozcan todos los comandos de un editor potente en particular. Tampoco esperamos que usted lo haga. Más bien sugerimos un enfoque más pragmático: aprenda los comandos que le faciliten la vida. La fórmula para esto es bastante simple.

Primero, fíjese en sí mismo mientras edita. Cada vez que se descubra haciendo algo repetitivo, acostúmbrese a pensar “debe de haber una manera mejor”. Después, encuéntrela.

Una vez que haya descubierto una funcionalidad nueva y útil, tiene que introducirla en su memoria muscular, de forma que pueda usarla sin pensar. El único modo que conocemos de hacer esto es a través de la repetición. Busque de manera consciente oportunidades para usar su nuevo superpoder, a ser posible muchas veces al día. Después de unas semanas, se dará cuenta de que lo usa sin pensar.

Hacer crecer su editor

La mayoría de los editores de código potentes se construyen en torno a un núcleo básico que después aumenta mediante extensiones. Muchas se proporcionan con el editor y otras pueden añadirse más adelante.

Cuando se tope con una aparente limitación del editor que está utilizando, busque una extensión que haga el trabajo. Hay muchas posibilidades de que no sea la única persona que necesita esa capacidad y, si tiene suerte, alguien habrá publicado una solución.

Lleve esto un paso más allá. Investigue a fondo el lenguaje de extensión de su editor. Averigüe cómo utilizarlo para automatizar algunas de las cosas repetitivas que hace. A menudo, solo necesitará una o dos líneas de código.

A veces, puede ir todavía más lejos y acabar escribiendo una extensión avanzada. Si eso ocurre, publíquela: si usted la ha necesitado, otras personas también lo harán.

Las secciones relacionadas incluyen

- Tema 7, “¡Comuníquese!”.

Retos

- Se acabó la repetición automática.
  Todo el mundo lo hace: necesita eliminar la última palabra que ha escrito, así que pulsa la tecla de retroceso y espera a que la repetición automática haga su trabajo. De hecho, estamos seguros de que su cerebro ha hecho esto tantas veces que puede saber casi con total exactitud cuándo dejar de pulsar la tecla. Así pues, desactive la repetición automática y, en vez de usarla, aprenda las secuencias de teclas para mover, seleccionar y eliminar por carácter, palabras, líneas y bloques.
- Esto va a doler.
  Desháguese del ratón/panel táctil. Durante una semana entera, edite utilizando solo el teclado. Se dará cuenta de que hay un montón de cosas que no sabe hacer sin apuntar y hacer clic, así que es hora de aprender. Tome notas (le recomendamos que lo haga a la antigua usanza, con lápiz y papel) de las secuencias de teclas que aprenda. Su productividad se verá afectada durante unos días, pero, a medida que aprenda a hacer cosas sin apartar las manos de la posición inicial,

verá que su edición es cada vez más rápida y fluida de lo que jamás había sido antes.

- Busque integraciones. Mientras escribíamos este capítulo, Dave se preguntó si podría ver una vista previa del diseño final (un archivo PDF) en un búfer de edición. Una descarga después, la maqueta está junto al texto original en el editor. Haga una lista de las cosas que le gustaría incorporar al editor y búsquelas.
- Siendo un poco más ambicioso, si no encuentra un *plugin* o extensión que haga lo que usted quiere, escriba uno. A Andy le gusta crear *plugins* Wiki personalizados y basados en archivos locales para sus editores favoritos. ¡Si no lo encuentra, créelo!

19 Control de versiones

El progreso, lejos de consistir en cambio, depende de la memoria. Los que no recuerdan el pasado están condenados a repetirlo.

—George Santayana, La vida de la razón.

Uno de los elementos importantes que buscamos en una interfaz de usuario es la opción para deshacer, un solo botón que perdona nuestros errores. Es mejor incluso si el entorno soporta múltiples niveles para deshacer y rehacer, de manera que podemos volver atrás y arreglar algo que ha sucedido hace un par de minutos.

Pero ¿qué pasa si el error se cometió la semana pasada y desde entonces hemos encendido y apagado el ordenador diez veces? Bueno, ese es uno de los muchos beneficios de utilizar un sistema de control de versiones: es un botón **Deshacer** gigante, una máquina del tiempo que abarca un proyecto completo y puede llevarnos de vuelta a esos días felices de la semana pasada, cuando el código sí se compilaba y se ejecutaba.

Para muchas personas, hasta ahí llega su uso del sistema de control de versiones. Esas personas están perdiéndose todo un mundo más grande de colaboración, *pipelines* de despliegue, seguimiento de problemas e interacción general con el equipo.

Así pues, vamos a echar un vistazo a los sistemas de control de versiones, primero como repositorio de cambios y, después, como punto de

encuentro central para su equipo y su código.

Los directorios compartidos NO son control de versiones

Todavía nos encontramos de vez en cuando con algún equipo que comparte los archivos fuente de sus proyectos a través de una red, ya sea de manera interna o utilizando algún tipo de almacenamiento en la nube. Eso no es viable.
Los miembros de los equipos que hacen eso están interfiriendo todo el tiempo en el trabajo de los demás, perdiendo cambios, rompiendo construcciones y liándose a puñetazos en el aparcamiento. Es como escribir código concurrente con datos compartidos y sin mecanismo de sincronización. Utilice el control de versiones.
¡Pero aún hay más! Algunas personas sí utilizan el control de versiones y mantienen su repositorio principal en una red o un servicio de alojamiento en la nube. Su razonamiento es que esto tiene lo mejor de los dos mundos: se puede acceder a los archivos desde cualquier parte y (en el caso del almacenamiento en la nube) hay una copia de seguridad externa.
Resulta que esto es aún peor y se arriesgan a perderlo todo. El software de control de versiones utiliza un conjunto de directorios y archivos que interactúan. Si dos instancias hacen cambios de manera simultánea, el estado general puede corromperse y no se puede predecir el daño que se causará. Y a nadie le gusta ver llorar a los desarrolladores.

Empieza en la fuente

Los sistemas de control de versiones llevan un registro de cada cambio que se hace en el código fuente y la documentación. Con un sistema de control del código fuente configurado de manera adecuada, siempre podemos volver a una versión anterior de nuestro software.

Pero un sistema de control de versiones sirve para mucho más que para deshacer errores. Un buen sistema de control de versiones permite rastrear los cambios, respondiendo a preguntas como: ¿quién ha hecho cambios en esta línea de código?, ¿cuál es la diferencia entre la versión actual y la de la semana pasada?, ¿cuántas líneas de código hemos cambiado en esta liberación? o ¿qué archivos hemos cambiado con mayor frecuencia? Este tipo de información tiene un valor incalculable para fines relacionados con el rastreo de fallos, las auditorías, el rendimiento y la calidad.

Un sistema de control de versiones también nos permite identificar liberaciones de nuestro software. Una vez identificada, siempre podremos volver atrás y regenerar la liberación, independientemente de los cambios que hayan podido realizarse después.

Los sistemas de control de versiones pueden guardar los archivos que mantienen en un repositorio central, un gran candidato para su archivo.

Por último, los sistemas de control de versiones permiten a dos o más usuarios trabajar en el mismo conjunto de archivos al mismo tiempo e incluso hacer cambios concurrentes en el mismo archivo. Después, el sistema gestiona la fusión de estos cambios cuando los archivos se envían de vuelta al repositorio. Aunque puede parecer arriesgado, estos sistemas funcionan bien en la práctica con proyectos de todos los tamaños.

Truco 28. Utilice siempre el control de versiones.

Siempre. Incluso si es un equipo de una sola persona trabajando en un proyecto de una semana. Incluso si es un prototipo "de usar y tirar". Incluso si aquello en lo que está trabajando no es el código fuente. Asegúrese de que todo está dentro del control de versiones: documentación, listados telefónicos, memorandos para los proveedores, *makefiles*, procedimientos de construcción y liberación, ese *script* pequeñito del intérprete de comandos que ordena los archivos de registro... Todo. Nosotros utilizamos de forma rutinaria un control de versiones de casi todo lo que escribimos (incluido este libro). Incluso si no estamos trabajando en un proyecto, nuestro trabajo diario está a buen recaudo en un repositorio.

Ramificaciones

Los sistemas de control de versiones no solo conservan un historial de nuestro proyecto. Una de sus características más potentes y útiles es la manera en que permiten aislar partes de desarrollo en elementos denominados "ramas". Podemos crear una rama en cualquier punto del historial de nuestro proyecto, y cualquier trabajo que realicemos en esa rama estará aislado de todas las demás ramas. En algún momento del futuro, podemos fusionar la rama en la que estamos trabajando con otra rama, de manera que la rama de destino ahora contiene los cambios que hemos hecho en nuestra rama. Incluso es posible que varias personas estén trabajando en una rama: en cierto modo, las ramas son como pequeños proyectos clónicos.

Un beneficio de las ramas es el aislamiento que proporcionan. Si desarrolla una funcionalidad A en una rama y otro miembro de su equipo trabaja en una funcionalidad B en otra, no van a interferir el uno con el otro.

Un segundo beneficio, que quizá le parezca sorprendente, es que a menudo, las ramas están en el corazón del flujo de trabajo de un proyecto.

Y aquí es donde las cosas se vuelven un poco confusas. Las ramas de control de versiones y la organización de pruebas tienen algo en común: en ambos casos hay miles de personas por ahí diciéndole cómo debería hacerlo. Y ese consejo es en su mayor parte carente de sentido, porque lo que en realidad están diciendo es “esto es lo que me funcionó a mí”.

Así pues, use el control de versiones en su proyecto y, si se topa con problemas relacionados con el flujo de trabajo, busque posibles soluciones. Y recuerde revisar y ajustar lo que está haciendo a medida que adquiere experiencia.

Un experimento de reflexión

Derrame una taza entera de té (típico inglés, con un poco de leche) sobre el teclado de su portátil. Lleve la máquina al bar de los listos y haga que chasqueen la lengua y frunzan el ceño. Compre un ordenador nuevo. Lléveselo a casa.
¿Cuánto tiempo tardaría en conseguir que esa máquina volviese al mismo estado en el que estaba (con todas las claves SSH, la configuración del editor, la configuración del intérprete de comandos, las aplicaciones instaladas, etc.) en el momento en que levantó aquella taza fatídica? Ese es un problema que uno de nosotros tuvo hace poco.
Prácticamente todo lo que definía la configuración y el uso del ordenador original estaba almacenado en el sistema de control de versiones, incluyendo:

- Todas las preferencias de usuario y *dotfiles*.
- La configuración del editor.
- La lista de software instalado mediante el uso de Homebrew.
- El *script* de Ansible utilizado para configurar aplicaciones.
- Todos los proyectos actuales.

La máquina ya se había restaurado a media tarde.

Control de versiones como centro de proyectos

Aunque el control de versiones resulta muy útil para proyectos personales, en realidad cobra más sentido cuando se trabaja en un equipo. Gran parte de este valor viene de la manera en que se aloja el repositorio.

Ahora, muchos sistemas de control de versiones no necesitan alojamiento. Están descentralizados por completo, con cada desarrollador cooperando de igual a igual. Pero, incluso con estos sistemas, merece la pena considerar tener un repositorio central, porque, una vez que lo tiene, puede beneficiarse de un montón de integraciones para hacer que el proyecto fluya con más facilidad.

Muchos de los sistemas de repositorio son de código abierto, así que puede instalarlos y ejecutarlos en su empresa. Pero esa no es realmente su línea de negocio, así que nosotros recomendamos a la mayoría de la gente que utilice alojamiento con terceros. Busque características como:

- Buena seguridad y control de acceso.
- Interfaz de usuario intuitiva.
- La capacidad para hacerlo todo desde la línea de comandos también (porque puede que necesite automatizarlo).
- Construcciones y pruebas automatizadas.
- Buen soporte para la fusión de ramas (que a veces se denomina *pull requests* o solicitudes de extracción).
- Gestión de problemas (lo ideal sería que se integrase en *commits* y fusiones, para poder mantener métricas).
- Buen sistema de informes (una pantalla de estilo tablero Kanban con tareas y problemas pendientes puede resultar muy útil).
- Buenas comunicaciones para los equipos: correos electrónicos u otras notificaciones sobre cambios, una wiki, etc.

Muchos equipos tienen sus sistemas de control de cambios configurados para que un *push* a una rama particular construya de forma automática el sistema, ejecute las pruebas y, si se estas se pasan, despliegue el nuevo código a producción.

¿Suena aterrador? No cuando se da cuenta de que está usando el control de versiones. Siempre puede volver a una versión anterior.

Las secciones relacionadas incluyen

- Tema 11, “Reversibilidad”.
- Tema 49, “Equipos pragmáticos”.

- Tema 51, “Kit pragmático básico”.

Retos

- Saber que puede volver a cualquier estado anterior utilizando el sistema de control de versiones es una cosa, pero ¿de verdad sabe hacerlo? ¿Conoce los comandos para hacerlo de forma adecuada? Apréndalos ahora, no cuando se produzca un desastre y se encuentre bajo presión.
- Dedique algo de tiempo a pensar en la recuperación del entorno de su propio portátil en caso de desastre. ¿Qué necesitaría recuperar? Muchas de las cosas que necesita son solo archivos de texto. Si no están en un sistema de control de versiones (alojadas fuera de su portátil), encuentre una manera de añadirlos. Después, piense en todo lo demás: aplicaciones instaladas, configuración del sistema, etc. ¿Cómo puede expresar todo eso en archivos de texto para que también pueda guardarse?
  Un experimento interesante, una vez que haya hecho progresos, es encontrar un ordenador viejo que ya no use y ver si su sistema nuevo puede utilizarse para configurarlo.
- Explore de manera consciente las características de su sistema de control de versión y su proveedor de alojamiento actuales que no está utilizando. Si su equipo no está usando ramas de características, experimente con su introducción. Haga lo mismo con las solicitudes de extracción/fusión. Integración continua. *Pipelines* de construcción. Incluso despliegue continuo. Fíjese también en las herramientas de comunicación del equipo: wikis, tablero Kanban, etc.
  No tiene por qué utilizar nada de eso, pero necesita saber qué hacen esas cosas para tomar esa decisión.
- Utilice el control de versiones también para cosas que no sean proyectos.

20 Depuración

Es algo doloroso

mirar tus propios problemas y saber
que tú mismo, y nadie más, los has provocado.

—Sófocles, Áyax.

La palabra *bug* (“bicho” en inglés) se usa para describir un “objeto de terror” desde el siglo XIV. A la contralmirante Dra. Grace Hopper, la inventora de COBOL, se le atribuye la observación del primer *bug* en un ordenador; literalmente, una polilla atrapada en un relé de un sistema informático primitivo. Cuando se pidió una explicación de por qué la máquina no estaba funcionando como debía, un técnico informó de que había “un bicho en el sistema” y lo pegó diligentemente, con alas y todo, en el cuaderno de bitácora.

Por desgracia, todavía tenemos *bugs* en el sistema, aunque no son voladores. Pero el significado del siglo XIV (algo aterrador) es quizá más aplicable ahora que entonces. Los defectos en el software se manifiestan de diversas maneras, desde requisitos mal entendidos a errores en el código. Por desgracia, los sistemas informáticos modernos todavía se limitan a hacer lo que les decimos que hagan, no necesariamente lo que queremos que hagan.

Nadie escribe software perfecto, así que es un hecho que la depuración ocupará una parte importante de nuestra jornada. Vamos a echar un vistazo a algunos de los problemas relacionados con la depuración y a algunas estrategias generales para encontrar *bugs* (fallos) esquivos.

Psicología de la depuración

La depuración es un tema sensible y emocional para muchos desarrolladores. En vez de enfrentarse a ella como un rompecabezas que hay que resolver, puede que se encuentre con negación, dedos acusatorios, malas excusas o simple apatía, sin más.

Acepte el hecho de que la depuración es solo la resolución de problemas y afróntela como tal.

Si encuentra un fallo de otra persona, puede invertir tiempo y energía en acusar al asqueroso culpable que lo ha creado. En algunos lugares de trabajo, esto es parte de la cultura y quizá sea catártico. Sin embargo, en el

escenario técnico, le conviene concentrarse en arreglar el problema, no en echar la culpa.

Truco 29. Arregle el problema, no busque culpables.

No importa si el fallo es culpa suya o de otra persona. Sigue siendo su problema.

Mentalidad de la depuración

La persona a la que se engaña con más facilidad es uno mismo.

—Edward Bulwer-Lytton, The Disowned.

Antes de empezar a depurar, es importante adoptar la mentalidad adecuada. Necesita desactivar muchas de las defensas que utiliza cada día para proteger su ego, desconectar de cualquier presión externa que pueda sufrir el proyecto y ponerse cómodo. Sobre todo, recuerde la primera regla de la depuración:

Truco 30. Que no cunda el pánico.

Es fácil entrar en un estado de pánico, sobre todo si tiene una fecha límite o un jefe o un cliente nervioso justo detrás de usted mientras intenta encontrar la causa del fallo. Pero es muy importante dar un paso atrás y pensar de verdad en lo que podría estar causando los síntomas que usted cree que indican la existencia de un fallo.

Si su primera reacción cuando presencia un fallo o ve un informe de fallos es decir: “Esto es imposible”, se equivoca. No desperdicie ni una neurona en un razonamiento que empieza por “pero eso no puede ocurrir”, porque puede, y ha ocurrido.

Tenga cuidado con la miopía cuando depure. Resista la tentación de arreglar solo los síntomas que ve: es más probable que el auténtico fallo esté muy alejado de lo que está observando y puede implicar otras cosas relacionadas. Intente siempre descubrir la raíz del problema, no solo este aspecto particular de él.

Por dónde empezar

Antes de empezar a fijarse en el fallo, asegúrese de que está trabajando en un código que se construyó de forma limpia, sin avisos. Nosotros, como rutina, configuramos los niveles de advertencia del compilador más altos posibles. ¡No tiene sentido perder tiempo intentando encontrar un problema que podría encontrar el ordenador por nosotros! Tenemos que concentrarnos en los problemas más difíciles que tenemos entre manos. Cuando intenta resolver cualquier problema, necesita reunir todos los datos relevantes. Por desgracia, los informes de fallos no son una ciencia exacta. Es fácil dejarse engañar por coincidencias y no puede permitirse malgastar el tiempo depurando coincidencias. Primero, necesita ser preciso en sus observaciones.

La exactitud en los informes de fallos disminuye más todavía cuando nos llegan a través de terceros; puede que tenga que ver al usuario que informó del fallo en acción para obtener un nivel suficiente de detalle.

Andy trabajó una vez en una aplicación de diseño gráfico grande. Cuando faltaba poco para la liberación, los encargados de las pruebas informaron de que la aplicación fallaba cada vez que hacían un trazo con un pincel concreto. El programador responsable afirmaba que no le pasaba nada; él había intentado pintar con ese pincel y funcionaba. Este diálogo duró varios días y los ánimos estaban cada vez más caldeados. Al final, los juntamos en la misma sala. El probador seleccionó el pincel e hizo un trazo desde la esquina superior derecha hasta la esquina inferior izquierda.

La aplicación explotó. “Uy,” dijo con voz débil el programador, que admitió entonces con vergüenza que solo había hecho trazos de prueba desde la parte inferior izquierda a la parte superior derecha, algo que no dejaba al descubierto el fallo.

Hay dos puntos clave en esta historia:

- Puede que necesite preguntar al usuario que ha informado del fallo para recopilar más datos de los que se le han dado al principio.
- Las pruebas artificiales (como la única pincelada del programador de la parte superior a la inferior) no comprueban lo suficiente de la aplicación. Debe poner a prueba de forma brutal condiciones de contorno y patrones de uso realistas de los usuarios finales. Debe

hacerlo de manera sistemática (consulte “Pruebas despiadadas y continuas” en el tema 51).

Estrategias de depuración

Una vez que crea que sabe lo que está pasando, es hora de averiguar lo que el programa cree que está pasando.

Reproducir fallos

No, nuestros fallos no van a multiplicarse de verdad (aunque algunos de ellos son lo bastante mayores para hacerlo de forma legal). Estamos hablando de un tipo de reproducción diferente.

La mejor manera de empezar a arreglar un fallo es hacer que sea reproducible. Al fin y al cabo, si no podemos reproducirlo, ¿cómo sabremos si se arregla? Pero queremos algo más que un fallo que pueda reproducirse al seguir una larga serie de pasos; queremos un fallo que pueda reproducirse con un solo comando. Es mucho más difícil arreglar un fallo si tenemos que realizar 15 pasos para llegar al punto en el que aparece dicho fallo. Así pues, esta es la regla más importante de la depuración:

Truco 31. Hay que fallar la prueba antes de arreglar el código.

A veces, al forzarse a aislar las circunstancias que muestran el fallo, obtendrá incluso una idea de cómo arreglarlo. El acto de escribir la prueba proporciona la esencia de la solución.

Programador en tierras extrañas

Toda esta charla acerca de aislar fallos está muy bien, pero al enfrentarse con 50.000 líneas de código trabajando contrarreloj, ¿qué puede hacer un pobre programador? Primero, fíjese en el problema. ¿Es un fallo total? Siempre nos sorprendemos cuando impartimos un curso que implica programación por la cantidad de desarrolladores que ven que aparece una excepción en rojo y se pasan de inmediato al código.

Truco 32. Lea el puñetero mensaje de error.

Nada más que decir.

Malos resultados

¿Qué pasa si no es un fallo total? ¿Qué pasa si es solo un mal resultado?

Póngase manos a la obra con un depurador y utilice la prueba fallida para desencadenar el problema.

Antes de nada, asegúrese de que también está viendo el valor incorrecto en el depurador. Nosotros dos hemos desperdiciado horas intentando rastrear un fallo solo para descubrir que esta ejecución concreta del código funcionaba bien.

A veces, el problema es evidente: `interest_rate` es `4.5` y debería ser `0.045`. Con más frecuencia, tendrá que investigar más para averiguar por qué el valor está mal, para empezar. Asegúrese de que sabe cómo subir y bajar por la pila de llamadas y examine el entorno de la pila local.

Consideramos que, a menudo, ayuda tener un boli y un papel a mano para poder tomar notas. En particular, muchas veces nos topamos con una pista y la seguimos solo para descubrir que no lleva a ninguna parte. Si no anotásemos dónde estábamos cuando empezamos a seguir la pista, podríamos perder mucho tiempo volviendo a ese punto.

A veces, estamos mirando un seguimiento de pila que parece que no se acaba nunca. En este caso, suele haber una manera más rápida de encontrar el problema que examinar todos y cada uno de los marcos de pila: utilizar una búsqueda binaria. Pero, antes de hablar de eso, vamos a echar un vistazo a otros dos escenarios con fallos habituales.

Sensibilidad a los valores de entrada

Esto lo ha vivido. Su programa funciona bien con todos los datos de prueba y sobrevive a la primera semana en producción con honores. Y entonces, de repente, falla al introducirle un conjunto de datos concreto.

Puede intentar mirar en el lugar en el que falla y trabajar hacia atrás, pero, a veces, es más fácil empezar con los datos. Consiga una copia del

conjunto de datos e introdúzcalo a través de una copia de la aplicación que se ejecute de manera local, asegurándose de que todavía falla. Después, haga una búsqueda binaria en los datos hasta que aísle exactamente qué valores de entrada están generando el fallo.

Regresiones a través de liberaciones

Está en un buen equipo y libera su software a producción. En algún punto aparece un fallo en código que funcionaba bien hace una semana. ¿No sería estupendo poder identificar el cambio específico que lo introdujo? ¿Sabe qué? Es el momento de la búsqueda binaria.

La búsqueda binaria

Todo graduado en ciencias de la computación se ha visto obligado a escribir código para una búsqueda binaria. La idea es simple. Estamos buscando un valor concreto en una matriz ordenada. Podríamos mirar cada valor uno a uno, pero acabaríamos mirando más o menos la mitad de las entradas por término medio hasta que encontrásemos el valor que queríamos o encontrásemos un valor mayor que él, lo que querría decir que el valor no está en la matriz. Pero es más útil utilizar el enfoque “divide y vencerás”. Elija un valor en la parte media de la matriz. Si es el valor que estaba buscando, pare. Si no, puede partir la matriz en dos. Si el valor que encuentra es mayor que el valor objetivo, entonces sabe que este debe estar en la primera mitad de la matriz y, de lo contrario, está en la segunda. Repita el procedimiento en la submatriz que corresponda y enseguida tendrá un resultado. (Como veremos en “Notación Big O”, en el tema 39, una búsqueda lineal es $O(n)$, y una búsqueda binaria es $O(\log n)$).

Así pues, la búsqueda binaria es mucho mucho más rápida en cualquier problema de un tamaño decente. Vamos a ver cómo aplicarla a la depuración.

Cuando se enfrente a un seguimiento de pila masivo y esté intentando averiguar exactamente qué función manipuló el valor en el error, puede hacer una búsqueda eligiendo un marco de pila en algún punto hacia la mitad y viendo si el error se manifiesta ahí. Si lo hace, entonces sabrá que

hay que concentrarse en los marcos anteriores; si no, será en los siguientes. Vuelva a buscar. Incluso aunque tenga 64 marcos en el seguimiento de pila, este enfoque le dará una respuesta después de seis intentos como mucho. Si se encuentra con fallos que aparecen en determinados conjuntos de datos, podría hacer lo mismo. Divida el conjunto de datos en dos para ver si el problema se produce al pasar una parte o la otra a la aplicación. Siga dividiendo los datos hasta obtener un conjunto de valores mínimo que exponga el problema.

Si su equipo ha introducido un fallo durante un conjunto de liberaciones, puede emplear el mismo tipo de técnica. Cree una prueba que haga fallar la liberación actual. A continuación, escoja una liberación en un punto intermedio entre el presente y la última versión que se sabe que funcionaba. Ejecute la prueba de nuevo y determine cómo afinar la búsqueda. Ser capaz de hacer esto es solo uno de los múltiples beneficios de contar con un buen control de versiones en sus proyectos. De hecho, muchos sistemas de control de versiones van más allá y automatizan el proceso, seleccionando liberaciones por nosotros en función del resultado de la prueba.

Registro y/o rastreo

Por lo general, los depuradores se centran en el estado del programa en este momento. A veces, necesitamos más; necesitamos ver el estado de un programa o una estructura de datos a lo largo del tiempo. Ver un seguimiento de pila solo puede decirnos cómo hemos llegado aquí directamente. Por lo general no puede decirnos qué estábamos haciendo antes de esta cadena de llamadas, sobre todo en sistemas basados en eventos.[2]

Las sentencias de rastreo son esos pequeños mensajes de diagnóstico que se imprimen en la pantalla o en un archivo y dicen cosas como “llegó aquí” y “valor de x = 2”. Es una técnica primitiva comparada con los depuradores de estilo IDE, pero es peculiarmente efectiva a la hora de diagnosticar varias clases de errores que los depuradores no pueden diagnosticar. El rastreo es muy valioso en cualquier sistema en el que el tiempo en sí mismo sea un factor: procesos concurrentes, sistemas en tiempo real y aplicaciones basadas en eventos.

Puede utilizar sentencias de rastreo para examinar a fondo el código. Es decir, puede añadir sentencias de rastreo a medida que desciende por el árbol de llamadas.

Los mensajes de rastreo deberían estar en un formato regular y coherente, ya que puede que quiera analizarlos sintácticamente de manera automática. Por ejemplo, si necesitase rastrear una fuga de recursos (como un desequilibrio entre aperturas y cierres en un archivo), podría rastrear cada `open` y cada `close` en un archivo de registro. Al procesar el archivo de registro con herramientas de procesamiento de texto o el intérprete de comandos, podrá identificar con facilidad dónde está la apertura que causa problemas.

Patito de goma

Una técnica muy sencilla, pero muy útil, para encontrar la causa de un problema es simplemente explicárselo a otros. La otra persona debería mirar la pantalla por encima de nuestro hombro y asentir todo el tiempo (como un patito de goma que se balancea arriba y abajo en la bañera). No hace falta que digan nada; el simple acto de explicar, paso a paso, lo que se supone que hace el código consigue a menudo que el problema salga de la pantalla y se presente solo ante nosotros.[3]

Suena muy simple, pero, al explicar el problema a otra persona, tenemos que decir de manera explícita cosas que quizá daríamos por sentadas al avanzar por el código nosotros solos. Cuando tenemos que verbalizar algunos de estos supuestos, de pronto conseguimos una nueva visión del problema. Y, si no hay otra persona cerca, un patito de goma, un osito de peluche o una planta en una maceta servirán.[4]

Proceso de eliminación

En la mayoría de proyectos, el código que tiene que depurar puede ser una mezcla de código de la aplicación escrito por usted y otros miembros del equipo del proyecto, productos de terceros (base de datos, conectividad, *framework* web, comunicaciones o algoritmos personalizados, etc.) y el

entorno de la plataforma (sistema operativo, bibliotecas del sistema, compiladores).

Es posible que exista un fallo en el SO, el compilador o un producto de terceros, pero eso no debería ser lo primero que se le pase por la cabeza. Es mucho más probable que el fallo exista en el código de la aplicación en desarrollo. Por lo general, es más productivo asumir que el código de la aplicación está llamando de manera incorrecta a una biblioteca que asumir que la biblioteca en sí está rota. Incluso si el problema está en un elemento de terceros, tendrá que eliminar su código antes de presentar el informe de fallos.

Trabajamos en un proyecto en el que un ingeniero sénior estaba convencido de que la llamada al sistema `select` estaba estropeada en un sistema Unix. No había persuasión ni lógica que le hiciese cambiar de idea (el hecho de que todas las demás aplicaciones de red de la caja funcionaban bien era irrelevante). Pasó semanas escribiendo soluciones alternativas que, por alguna extraña razón, no arreglaban el problema. Cuando por fin se vio obligado a sentarse y leer la documentación de `select`, descubrió el problema y lo corrigió en solo unos minutos. Ahora utilizamos la frase "select está estropeado" como recordatorio amable cada vez que uno de nosotros empieza a echar la culpa al sistema por un fallo que probablemente es nuestro.

Truco 33. "select" no está estropeado.

Recuerde, si ve huellas de cascos, piense en caballos, no en cebras. Es probable que el SO no esté estropeado. Y es probable que a `select` no le pase nada.

Si "solo ha cambiado una cosa" y el sistema ha dejado de funcionar, hay muchas probabilidades de que esa única cosa sea la responsable, de forma directa o indirecta, no importa lo disparatado que parezca. A veces, la cosa que cambia está fuera de nuestro control: versiones nuevas del SO, el compilador, la base de datos u otro software de terceros pueden sembrar el caos en un código que antes era correcto. Podrían aparecer fallos nuevos. Fallos para los que tenía una solución alternativa se reparan, estropeando la solución alternativa. Las API cambian, la funcionalidad cambia; en

resumen, es una partida totalmente nueva, y debe volver a probar el sistema bajo esas condiciones nuevas. Por tanto, no pierda de vista el calendario cuando se plantee una mejora; quizá prefiera esperar hasta después de la siguiente liberación.

El elemento sorpresa

Cuando le parezca que un fallo es sorprendente (quizá incluso murmurando “eso es imposible” muy bajito para que no le oigamos), debe reevaluar todas las verdades en las que cree. En ese algoritmo para calcular descuentos, ese que sabía que era a prueba de bombas y que sería imposible que fuese la causa de este fallo, ¿probó todas las condiciones de contorno? Esa otra porción de código que lleva años usando no puede tener un fallo. ¿Verdad?

Por supuesto que puede. El grado de sorpresa que siente cuando algo va mal es proporcional al grado de confianza y fe que tiene en el código que está ejecutándose. Esa es la razón por la que, cuando se enfrente a un fallo “sorprendente”, debe aceptar que una o más de sus asunciones está mal. No repase por encima una rutina o porción de código involucrada en el fallo porque “sabe” que funciona. Demuéstrelo. Demuéstrelo en este contexto, con estos datos, con estas condiciones de contorno.

Truco 34. No lo dé por sentado; demuéstrelo.

Cuando se tope con un fallo sorpresa, más allá de arreglarlo sin más, tiene que determinar por qué ese fallo no se ha detectado antes. Plantéese si necesita modificar las pruebas unitarias u otras para que lo hubiesen detectado.

Además, si el fallo es el resultado de datos malos que se han propagado por un par de niveles antes de causar la explosión, piense si una mejor comprobación de parámetros en esas rutinas lo habrían aislado antes (consulte las explicaciones sobre fallos totales tempranos y aserciones en el tema 24 y el tema 25, respectivamente).

Ya que está en ello, ¿hay otras partes del código que podrían ser susceptibles del mismo fallo? Ahora es el momento de encontrarlas y

arreglarlas. Asegúrese de que, haya pasado lo que haya pasado, lo sabrá si vuelve a ocurrir.

Si ha tardado mucho tiempo en arreglar este fallo, pregúntese por qué. ¿Hay algo que pueda hacer para que sea más fácil arreglar este fallo la próxima vez? Quizá podría incorporar mejores enganches de pruebas o escribir un analizador de archivos de registro.

Por último, si el fallo es el resultado de una asunción equivocada por parte de otra persona, hable del problema con todo el equipo: si una persona lo ha entendido mal, es posible que no sea la única.

Si hace todo esto, con un poco de suerte no se llevará una sorpresa la próxima vez.

Lista de comprobación de depuración

- El problema del que está informándose ¿es un resultado directo del fallo subyacente o solo un síntoma?
- ¿Está de verdad el fallo en el *framework* que está utilizando? ¿Está en el SO? ¿O está en su código?
- Si tuviese que explicar este problema con detalle a un compañero, ¿qué le diría?
- Si el código sospechoso pasa sus pruebas unitarias, ¿son las pruebas lo bastante completas? ¿Qué ocurre si ejecuta las pruebas con estos datos?
- Las condiciones que han causado este fallo ¿existen en alguna otra parte del sistema? ¿Hay otros fallos todavía en fase larvaria, esperando para salir del cascarón?

Las secciones relacionadas incluyen

- Tema 24, “Los programas muertos no mienten”.

Retos

- La depuración ya es un buen reto de por sí.

21 Manipulación de texto

Los programadores pragmáticos manipulan texto de la misma manera que los carpinteros dan forma a la madera. En secciones anteriores hemos hablado de algunas herramientas específicas (intérpretes de comandos, editores, depuradores) que utilizamos. Son similares a los escoplos, las sierras y las garlopas, herramientas especializadas para hacer una o dos tareas bien. Sin embargo, de vez en cuando, necesitamos realizar algún tipo de transformación que no puede manejarse con facilidad con el conjunto de herramientas básico. Necesitamos una herramienta de manipulación de texto de uso general.

Los lenguajes de manipulación de texto son a la programación lo que las rebajadoras son a la carpintería. Hacen ruido, generan desorden y, en cierto modo, utilizan la fuerza bruta. Si comete un error con ellos, se puede arruinar un trabajo completo. Pero, en las manos adecuadas, tanto las rebajadoras como los lenguajes de manipulación de texto pueden ser muy potentes y versátiles. Puede recortar algo con rapidez para darle forma, crear juntas y tallar. Si se utilizan de modo adecuado, estas herramientas tienen una finura y una delicadeza sorprendentes. Pero hace falta tiempo para dominarlas.

Por suerte, hay varios lenguajes de manipulación de texto estupendos. A los desarrolladores de Unix (y aquí incluimos a los usuarios de macOS) con frecuencia les gusta utilizar el poder de sus intérpretes de comandos, aumentado con herramientas como `awk` y `sed`. A las personas que prefieran una herramienta más estructurada puede que les gusten más lenguajes como Python o Ruby.

Estos lenguajes son tecnologías propicias importantes. Al utilizarlas, puede modificar con rapidez utilidades e ideas para prototipos, tareas que tardarían cinco o diez veces más en hacerse usando lenguajes convencionales. Y ese factor de multiplicación es importantísimo para el tipo de experimentos que realizamos. Pasar 30 minutos probando una idea loca es mucho mejor que pasar cinco horas. Dedicar un día a automatizar componentes importantes de un proyecto es aceptable; dedicar una semana podría no serlo. En su libro *La práctica de la programación* [KP99], Kernighan y Pike creaban el mismo programa en cinco lenguajes diferentes.

La versión en Perl era la más corta (17 líneas, comparadas con las 150 de C). Con Perl podemos manipular texto, interactuar con programas, hablar a través de redes, controlar páginas web, usar aritmética de precisión arbitraria y escribir programas que parezcan Snoopy diciendo palabrotas.

Truco 35. Aprenda un lenguaje de manipulación de texto.

Para mostrar la aplicabilidad tan amplia de los lenguajes de manipulación de texto, vamos a ver una muestra de algunas cosas que hemos hecho con Ruby y Python relacionadas solo con la creación de este libro:

- **Crear el libro:** El sistema de creación para Pragmatic Bookshelf está escrito en Ruby. Autores, editores, maquetadores y personal de soporte utilizan tareas de Rake para coordinar la creación de PDF y libros electrónicos.
- **Inclusión de código y resaltado:** Creemos que es importante que cualquier código que se presente en un libro se haya probado primero. La mayoría del código de este libro se ha probado. Sin embargo, siguiendo el principio DRY (consulte el tema 9, “DRY: los males de la duplicación”), no queríamos copiar y pegar líneas de código de los programas probados al libro. Eso supondría duplicar código, garantizando prácticamente que olvidaríamos actualizar un ejemplo cuando se modificase el programa correspondiente. Para algunos ejemplos, tampoco queríamos aburrirle con todo el código de *framework* necesario para hacer que nuestro ejemplo se ejecutase y compilase. Utilizamos Ruby. Se invoca un *script* relativamente simple cuando aplicamos formato al libro; extrae un segmento con nombre de un archivo fuente, hace un resaltado de sintaxis y convierte el resultado en el lenguaje de composición tipográfica que utilizamos.
- **Actualización del sitio web:** Tenemos un *script* simple que realiza una construcción parcial del libro, extrae la tabla de contenido y la sube a la página del libro en nuestro sitio web. También tenemos un *script* que extrae secciones de un libro y las sube como muestras.

- **Incluir ecuaciones:** Hay un *script* en Python que convierte marcado matemático LaTeX en texto con un formato bonito.
- **Generación de índices:** La mayoría de los índices se crean como documentos separados (lo que hace que su mantenimiento sea difícil si un documento cambia). Los nuestros están marcados en el propio texto y un *script* de Ruby ordena y da formato a las entradas.

Y así sucesivamente. De una manera muy real, Pragmatic Bookshelf se construye en torno a la manipulación de texto. Si sigue nuestro consejo de mantener las cosas en texto simple, utilizar estos lenguajes para manipular ese texto le reportará numerosos beneficios.

Las secciones relacionadas incluyen

- Tema 16, “El poder del texto simple”.
- Tema 17, “Jugar con el intérprete de comandos”.

Ejercicios

Ejercicio 11

Va a escribir una aplicación que solía utilizar YAML como lenguaje de configuración. Ahora su empresa ha adoptado JSON como estándar, así que tiene un montón de archivos `.yaml` que tiene que convertir a `.json`. Escriba un *script* que tome un directorio y convierta cada archivo `.yaml` en un archivo `.json` correspondiente (de forma que `database.yaml` se convierta en `database.json`, y los contenidos sean válidos en JSON).

Ejercicio 12

Al principio, su equipo eligió utilizar nombres `camelCase` para variables, pero después cambió de idea y pasó a `snake_case`. Escriba un *script* que examine todos los archivos fuente en busca de nombres camelCase e informe de ellos.

Ejercicio 13

Como continuación del ejercicio anterior, añada la capacidad para cambiar esos nombres de variables de manera automática en uno o más archivos. Recuerde conservar una copia de seguridad de los originales por si algo sale muy muy mal.

22 Cuadernos de bitácora de ingeniería

Dave trabajó una vez para un pequeño fabricante de ordenadores, lo que supuso trabajar con ingenieros electrónicos y, a veces, mecánicos. Muchos de ellos se paseaban por ahí con un cuaderno de papel, por lo general con un boli metido en la espiral. A veces, cuando estaban hablando, abrían el cuaderno y escribían algo.

Al final, Dave hizo la pregunta evidente. Resultó que les habían formado para llevar "cuadernos de bitácora" de ingeniería, una especie de diario en el que registraban lo que hacían, cosas que habían aprendido, bocetos de ideas, lecturas de medidores: básicamente, cualquier cosa que tuviese que ver con su trabajo. Cuando se llenaba el cuaderno, escribían el rango de fechas en el lateral y lo dejaban en la estantería junto a los cuadernos anteriores. Puede que hubiese cierta competencia por ver quién tenía más cuadernos en la estantería.

Nosotros utilizamos cuadernos de bitácora para tomar notas en reuniones, apuntar en qué estamos trabajando, anotar valores variables al depurar, escribir recordatorios de dónde ponemos las cosas, registrar ideas salvajes y, a veces, solo para hacer garabatos.[5] El cuaderno de bitácora tiene tres beneficios principales:

- Es más fiable que la memoria. Alguien puede preguntarle: "¿Cómo se llamaba esa empresa a la que llamaste por el problema con el suministro eléctrico?" y podemos volver a la página de antes y darle el nombre y el número.
- Nos ofrece un lugar para almacenar ideas que no son relevantes en ese momento para la tarea que tenemos entre manos. De ese modo, podemos seguir concentrándonos en lo que estamos haciendo, sabiendo que no vamos a olvidar esa idea genial.

- Actúa como una especie de patito de goma (que ya hemos descrito). Cuando nos detenemos para apuntar algo, nuestro cerebro puede cambiar de marcha, casi como si hablásemos con otra persona; es una gran oportunidad para reflexionar. Puede que empecemos escribiendo una nota y, de repente, nos demos cuenta de que lo que acabamos de hacer, el tema de la nota, está equivocado.

Hay también otro beneficio adicional. De vez en cuando, podemos echar la vista atrás para saber lo que estábamos haciendo hace muchos años y pensar en las personas, los proyectos y la ropa y los peinados horrorosos.

Así pues, intente llevar un cuaderno de bitácora de ingeniería. Utilice papel, no un archivo o una wiki: hay algo especial en el acto de escribir a mano en comparación con teclear. Dese un plazo de un mes y compruebe si obtiene algún beneficio.

Al menos le ayudará a escribir sus memorias con más facilidad cuando sea rico y famoso, incluso si no le sirve para nada más.

Las secciones relacionadas incluyen

- Tema 6, “Su cartera de conocimientos”.
- Tema 37, “Escuche a su cerebro reptiliano”.

---

[1] Todo software se convierte en software heredado en cuanto se escribe.

[2] Aunque el lenguaje Elm tiene un depurador que viaja en el tiempo.

[3] ¿Por qué "patito de goma"? Mientras estudiaba en el Imperial College de Londres, Dave trabajó mucho con un asistente de investigación llamado Greg Pugh, uno de los mejores desarrolladores que Dave ha conocido. Durante varios meses, Greg llevó consigo un patito de goma amarillo que ponía al lado de su terminal mientras escribía código. Pasó un tiempo hasta que Dave se atrevió a preguntar...

[4] Las primeras versiones del libro mencionaban hablarle a una planta de marihuana. Era una errata. De verdad.

[5] Hay evidencias de que hacer garabatos ayuda a concentrarse y mejora las habilidades cognitivas. Puede ver un ejemplo en *What does doodling do?* [And10].

4
Paranoia pragmática

Truco 36. No puede escribir software perfecto.

¿Le ha dolido? No debería. Acéptelo como un axioma de la vida. Abrácelo. Celébrelo. Porque el software perfecto no existe. Nadie en la breve historia de la informática ha escrito un software perfecto. Es poco probable que usted sea el primero y, si no acepta esto como un hecho, malgastará tiempo y energía persiguiendo un sueño imposible.

Así pues, teniendo en cuenta esta realidad deprimente, ¿cómo convierte esto en una ventaja un programador pragmático? Ese es el tema de este capítulo.

Cada persona sabe que es, personalmente, la única buena conductora de la Tierra. El resto del mundo está por ahí para fastidiarles, saltándose las señales de stop, zigzagueando entre los carriles, sin usar el intermitente y, en general, sin cumplir nuestros estándares. Así que conducimos a la defensiva. Buscamos los problemas antes de que ocurran, nos anticipamos a lo inesperado y nunca nos ponemos en una situación de la que no podamos salir.

La analogía con la creación de código es bastante obvia. Estamos interactuando todo el tiempo con el código de otras personas (código que quizá no se ajuste a nuestros estándares elevados) y tratando con entradas que pueden ser o no ser válidas, así que se nos enseña a escribir código a la defensiva. Si hay alguna duda, validamos toda la información que se nos da. Usamos aserciones para detectar datos malos y desconfiamos de los datos que llegan de posibles atacantes o troles. Comprobamos si hay consistencia, ponemos restricciones en las columnas de la base de datos y, en general, nos sentimos bastante bien con nosotros mismos.

Pero los programadores pragmáticos van un paso más allá. Tampoco confían en sí mismos. Como saben que nadie escribe código perfecto,

incluidos ellos, los programadores pragmáticos construyen defensas contra sus propios errores. Describimos la primera medida defensiva en “Diseño por contrato”: clientes y proveedores deben estar de acuerdo con los derechos y las responsabilidades.

En “Los programas muertos no mienten”, creemos que no causamos daños mientras arreglamos los fallos, así que intentamos comprobar las cosas con frecuencia y finalizamos el programa si algo va mal.

“Programación asertiva” describe un método sencillo para ir comprobando sobre la marcha, escribir código que verifique de manera activa nuestros supuestos.

A medida que nuestros programas se vuelvan más dinámicos, se encontrará haciendo malabares con los recursos del sistema (memoria, archivos, dispositivos, etc.). En “Cómo equilibrar los recursos”, sugerimos maneras de garantizar que no se le cae ninguna de las mazas.

Y, lo que es más importante, nos ceñimos siempre a los pasos pequeños, como se describe en “No vaya más rápido que sus faros”, para no caernos por el precipicio.

En un mundo de sistemas imperfectos, plazos ridículos, herramientas irrisorias y requisitos imposibles, vamos a ir sobre seguro. Como dijo Woody Allen: “Cuando todo el mundo va a por ti, la paranoia es solo una buena forma de pensar”.

23 Diseño por contrato

Nada asombra tanto a los hombres como el sentido común y el trato sencillo.

—Ralph Waldo Emerson, Ensayos.

Tratar con sistemas informáticos es difícil. Tratar con personas es aún más difícil. Pero, como especie, ya no tenemos que averiguar cómo funcionan las interacciones humanas. Algunas de las soluciones que se nos han ocurrido en los últimos milenios pueden aplicarse también a la escritura de software. Una de las mejores soluciones para garantizar un trato sencillo es el contrato. Un contrato define nuestros derechos y responsabilidades, así como los de la otra parte. Además, hay un acuerdo respecto a las repercusiones si alguna de las partes no cumple el contrato.

Puede que tenga un contrato laboral que especifique las horas que trabajará y las normas de conducta que debe cumplir. A cambio, la empresa le paga un salario y otras gratificaciones. Cada parte cumple sus obligaciones y todo el mundo se beneficia.

Es una idea utilizada en todo el mundo, tanto de manera formal como informal, para ayudar a los humanos a interactuar. ¿Podemos utilizar el mismo concepto para ayudar a los módulos de software a interactuar? La respuesta es "sí".

DBC

Bertrand Meyer (*Construcción de software orientado a objetos* [Mey97]) desarrolló el concepto de "diseño por contrato" para el lenguaje Eiffel.[1] Se trata de una técnica sencilla, pero potente, que se centra en documentar (y estar de acuerdo con) los derechos y responsabilidades de los módulos de software para garantizar que el programa es correcto. ¿Qué es un programa correcto? Uno que no hace ni más ni menos de lo que afirma que hace. Documentar y verificar esa afirmación es el núcleo del diseño por contrato (DBC, por sus siglas en inglés).

Cada función y método en un sistema de software hace algo. Antes de comenzar ese algo, esa función puede tener algunas expectativas sobre el estado del mundo y puede que sea capaz de hacer una declaración sobre el estado del mundo cuando acabe. Meyer describe estas expectativas y afirma lo siguiente:

- **Precondiciones:** Lo que debe ser cierto para que se llame a la rutina; los requisitos de la rutina. Nunca debería llamarse a una rutina cuando se violarían sus precondiciones. Es responsabilidad del cliente pasar datos buenos.
- **Postcondiciones:** Lo que está garantizado que haga la rutina; el estado del mundo cuando la rutina acaba. El hecho de que la rutina tenga una postcondición implica que terminará: los bucles infinitos no están permitidos.
- **Invariantes de clase:** Una clase garantiza que esta condición siempre es cierta desde la perspectiva de un cliente. Durante el

procesamiento interno de una rutina, la invariante puede no cumplirse, pero, para cuando la rutina salga y el control vuelva al cliente, la invariante debe ser verdadera. (Tenga en cuenta que una clase no puede dar acceso de escritura sin restricciones a cualquier miembro de los datos que participe en la invariante).

Así, el contrato entre una rutina y cualquier cliente potencial puede leerse de este modo:

> *Si el cliente cumple todas las precondiciones de la rutina, la rutina garantizará que todas las postcondiciones e invariantes serán verdaderas cuando se complete.*

Si cualquier parte incumple los términos del contrato, entonces se invoca una solución (que se ha acordado con anterioridad), quizá se lanza una excepción o el programa termina. Pase lo que pase, debe tener claro que no lograr cumplir el contrato es un fallo. Es algo que nunca debería ocurrir, razón por la cual las precondiciones no deberían utilizarse para llevar a cabo cosas como la validación de entradas de usuario.

Algunos lenguajes soportan mejor estos conceptos que otros. Clojure, por ejemplo, soporta precondiciones y postcondiciones, además de una instrumentación extensa proporcionada por las especificaciones. Veamos un ejemplo de una función de banca para hacer un depósito utilizando precondiciones y postcondiciones simples:

```
(defn accept-deposit [account-id amount]
  { :pre [ (> amount 0.00)
       (account-open? account-id) ]
    :post [ (contains? (account-transactions account-id) %) ] }
  “Accept a deposit and return the new transaction id”
  ;; Aquí hay algo de procesamiento...
  ;; Devuelve la transacción recién creada:
  (create-transaction account-id :deposit amount))
```

Hay dos precondiciones para la función `accept-deposit`. La primera es que la cantidad es mayor que cero y la segunda es que la cuenta está abierta y es válida, como determina una función llamada `account-open?`. También hay una postcondición: la función garantiza que la nueva transacción (el valor de retorno de esta función, representado aquí con “%”) puede encontrarse entre las transacciones para esta cuenta.

Si llama a `accept-deposit` con una cantidad positiva para el depósito y una cuenta válida, procederá a crear una transacción del tipo apropiado y a hacer cualquier otro procesamiento que tenga que hacer. Sin embargo, si hay un fallo en el programa y, de algún modo, pasa una cantidad negativa para el depósito, recibirá una excepción de tiempo de ejecución:

```
Exception in thread “main”...
Caused by: java.lang.AssertionError: Assert failed: (> amount 0.0)
```

De manera similar, esta función requiere que la cuenta especificada esté abierta y sea válida. Si no es así, verá esta excepción en vez de eso:

```
Exception in thread “main”...
Caused by: java.lang.AssertionError: Assert failed: (account-open?
account-id)
```

Otros lenguajes tienen características que, aunque no son específicas del DBC, también pueden usarse con buenos resultados. Por ejemplo, Elixir utiliza cláusulas de guarda para enviar llamadas a funciones frente a varios cuerpos disponibles:

```
defmodule Deposits do
  def accept_deposit(account_id, amount) when (amount > 100000) do
    # ¡Llame al director!
  end
  def accept_deposit(account_id, amount) when (amount > 10000) do
    # Requisitos federales adicionales para creación de informes
    # Algo de procesamiento...
  end
  def accept_deposit(account_id, amount) when (amount > 0) do
    # Algo de procesamiento...
  end
end
```

En este caso, llamar a `accept_deposit` con una cantidad lo bastante grande puede desencadenar procesamiento y pasos adicionales. Sin embargo, si intenta llamarla con una cantidad inferior o igual a cero, recibirá una excepción informándole de que no puede:

```
** (FunctionClauseError) no function clause matching in
Deposits.accept_deposit/2
```

Este enfoque es mejor que comprobar las entradas sin más; en este caso, no puede simplemente llamar a esta función si sus argumentos están fuera de un rango.

Truco 37. Diseñe con contratos.

En “Ortogonalidad” recomendábamos escribir código “tímido”. Aquí, el énfasis está en código “perezoso”: sea estricto respecto a lo que va a aceptar antes de empezar y prometa lo menos posible a cambio. Recuerde, si su contrato indica que aceptará cualquier cosa y promete el mundo a cambio, ¡va a tener que escribir mucho código!

En cualquier lenguaje de programación, ya sea funcional, orientado a objetos o por procedimientos, el DBC nos obliga a pensar.

Invariantes de clase y lenguajes funcionales

Es una cuestión de nombres. Eiffel es un lenguaje orientado a objetos, así que Meyer llamó a esta idea “invariante de clase”. Pero, en realidad, es más general que eso. A lo que se refiere esta idea en realidad es al estado. En un lenguaje orientado a objetos, el estado se asocia a instancias de clases. Pero otros lenguajes también tienen estado.

En un lenguaje funcional, por lo general pasamos el estado a las funciones y recibimos un estado actualizado como resultado. Los conceptos de invariantes son igual de útiles en estas circunstancias.

DBC y desarrollo guiado por pruebas

¿Se necesita el diseño por contrato en un mundo en el que los desarrolladores practican pruebas unitarias, desarrollo guiado por pruebas (*test-driven development*, TDD), pruebas basadas en propiedades o programación defensiva? La respuesta corta es “sí”.

El DBC y la realización de pruebas son enfoques distintos para el tema más alto de los programas correctos. Ambas cosas tienen valor y ambas tienen sus usos en situaciones diferentes. El DBC ofrece varias ventajas sobre enfoques específicos de la realización de pruebas:

- El DBC no requiere configuración ni uso de ningún tipo de *mock*.

- El DBC define los parámetros para el éxito o el fracaso en todos los casos, mientras que las pruebas solo pueden tener como objetivo un caso específico cada vez.
- El TDD y otras pruebas solo se producen en el “tiempo de prueba” dentro del ciclo de construcción, pero el DBC y las aserciones son para siempre: durante el diseño, el desarrollo, el despliegue y el mantenimiento.
- El TDD no se centra en comprobar invariantes internas dentro del código que está probándose, es más de estilo caja negra para comprobar la interfaz pública.
- El DBC es más eficiente (y más adecuado al principio DRY) que la programación defensiva, en la que todo el mundo tiene que validar datos por si no lo hace nadie más.

El TDD es una técnica estupenda, pero, como ocurre con muchas técnicas, podría invitarle a concentrarse en el “camino feliz” y no en el mundo real llenos de datos malos, actores malos, versiones malas y especificaciones malas.

Implementar DBC

La simple enumeración de cuál es el rango del dominio de entrada, cuáles son las condiciones de contorno y qué promete entregar la rutina (o, lo que es más importante, qué no promete entregar) antes de escribir el código es un gran salto hacia delante para escribir un software mejor. Si no deja claras estas cosas, volverá a la programación por casualidad, que es donde muchos proyectos comienzan, terminan y fracasan.

En lenguajes que no soportan DBC en el código, quizá no pueda llegar más allá de ahí, y eso no es demasiado malo. Al fin y al cabo, el DBC es una técnica de diseño. Incluso sin comprobación automática, puede poner el contrato en el código como comentarios o en las pruebas unitarias y obtener un beneficio muy real.

Aserciones

Aunque documentar estos supuestos es un buen comienzo, podemos obtener un beneficio mucho mayor si hacemos que el compilador compruebe el contrato por nosotros. Esto se puede emular de manera parcial en algunos lenguajes mediante el uso de aserciones: comprobaciones en tiempo de ejecución de condiciones lógicas (consulte el tema 25,

“Programación asertiva”). ¿Por qué solo de manera parcial? ¿No podemos usar aserciones para hacer todo lo que el DBC puede hacer?

Por desgracia, la respuesta es no. Para empezar, en lenguajes orientados a objetos es probable que no haya soporte para propagar aserciones hacia abajo en una jerarquía de herencia. Eso significa que, si sobrescribe un método de clase base que tiene un contrato, no se llamará correctamente a las aserciones que implementan ese contrato (a menos que las duplique a mano en el nuevo código). Debe recordar llamar a la invariante de clase (y a todas las invariantes de clase base) a mano antes de salir de cada método. El problema básico es que el contrato no se cumple de manera automática.

En otros entornos, las excepciones generadas desde las aserciones de estilo DBC podrían desactivarse a nivel global o ignorarse en el código.

Además, no hay un concepto integrado de valores “antiguos”, es decir, valores como existían en la entrada a un método. Si está utilizando aserciones para cumplir contratos, debe añadir código a la precondición para guardar cualquier información que vaya a querer usar en la postcondición, si el lenguaje lo permite. En el lenguaje Eiffel, donde nació el DBC, puede utilizar simplemente la expresión `old`.

Por último, las bibliotecas y los sistemas en tiempo de ejecución convencionales no están diseñados para soportar contratos, así que estas llamadas no se comprueban. Es una gran pérdida, porque a menudo la mayoría de los problemas se detectan en el límite entre el código y las bibliotecas que utiliza (consulte el tema 24, “Los programas muertos no mienten”, para ver una explicación más detallada).

DBC y fallo total temprano

El DBC encaja bien con nuestro concepto del fallo total temprano (consulte el tema 24, “Los programas muertos no mienten”). Al utilizar un mecanismo de DBC o aserción para validar las precondiciones, postcondiciones e invariantes, podemos tener un fallo total temprano y ofrecer información más exacta sobre el problema.

Por ejemplo, supongamos que tenemos un método que calcula raíces cuadradas. Necesita una precondición de DBC que restrinja el dominio a números positivos. En lenguajes compatibles con el DBC, si pasamos a

`sqrt` un parámetro negativo, recibiremos un error informativo, como `sqrt_arg_must_be_positive`, junto con un seguimiento de pila.

Esto es mejor que la alternativa en otros lenguajes como Java, C y C++, en los que pasar un número negativo a `sqrt` devuelve el valor especial `NaN` (*Not a Number*, no es un número). Puede que más adelante en el programa intente hacer algunos cálculos con `NaN`, con resultados sorprendentes.

¿Quién es responsable?

¿Quién es responsable de comprobar la precondición: el cliente o la rutina a la que se llama? Cuando se implementa como parte del lenguaje, la respuesta es ninguno: la precondición se prueba entre bastidores después de que el cliente invoque la rutina, pero antes de que la rutina en sí se introduzca. Así, si hay que hacer una comprobación explícita de parámetros, debe realizarla el cliente, porque la rutina en sí nunca verá parámetros que violen su precondición. (Para lenguajes sin soporte integrado, necesitaríamos encerrar la rutina llamada con un preámbulo o postámbulo que compruebe estas aserciones). Piense en un programa que lea un número desde la consola, calcule su raíz cuadrada (llamando a `sqrt`) e imprima el resultado. La función `sqrt` tiene una precondición: su argumento no debe ser negativo. Si el usuario introduce un número negativo en la consola, depende del código que hace la llamada asegurarse de que nunca se pasa a `sqrt`. Este código tiene muchas opciones: podría finalizar, podría emitir una alerta y leer otro número o podría convertir el número en positivo y añadir una *i* al resultado devuelto por `sqrt`. Sea cual sea su elección, está claro que no es problema de `sqrt`.
Al expresar el dominio de la función de la raíz cuadrada en la precondición de la rutina `sqrt`, pasamos la responsabilidad de la corrección al cliente, donde pertenece. Entonces, podemos diseñar la rutina `sqrt` con la seguridad de saber que su entrada estará dentro del rango.

Es mucho más fácil encontrar y diagnosticar el problema si el fallo se produce pronto, en el lugar del problema.

Invariantes semánticas

Podemos utilizar invariantes semánticas para expresar requisitos inviolables, una especie de “contrato filosófico”. Una vez escribimos un *switch* de transacciones para tarjetas de débito. Un requisito importante era

que al usuario de la tarjeta de débito nunca se le debería aplicar la misma transacción a su cuenta dos veces. Dicho de otro modo, daba igual qué tipo de modo de fallo se produjese, el error debería inclinarse por no procesar una transacción en vez de procesar una transacción duplicada.

Esa norma sencilla, controlada directamente por los requisitos, resultó ser muy útil a la hora de resolver situaciones de recuperación de errores complejas y fue una gran orientación para el diseño y la implementación detallados en muchas áreas.

Asegúrese de no confundir requisitos que son normas fijas e inviolables con aquellos que son simples políticas que podrían cambiar con un nuevo régimen de gestión. Por eso utilizamos el término “invariantes semánticas”; debe ser primordial para el propio significado de una cosa, y no estar sujeto a los caprichos de las políticas (que es para lo que están las reglas de negocio, más dinámicas).

Cuando encuentre un requisito que entre en esta categoría, asegúrese de que se convierte en una parte conocida de cualquier documentación que esté produciendo, ya sea una lista con boliches en el documento de requisitos que se firma por triplicado o solo una nota grande en la pizarra común que ve todo el mundo. Intente expresarla de manera clara y sin ambigüedad. Por ejemplo, en el ejemplo de la tarjeta de débito, se podría escribir:

> *Errar en favor del cliente.*

Esta es una frase clara, concisa y sin ambigüedad que es aplicable en muchas áreas diferentes del sistema. Es nuestro contrato con todos los usuarios del sistema, nuestra garantía de comportamiento.

Contratos dinámicos y agentes

Hasta ahora, hemos hablado de los contratos como especificaciones fijas e inmutables, pero, en el paisaje de los agentes autónomos, ese no tiene por qué ser el caso. Según la definición de “autónomos”, los agentes son libres de rechazar solicitudes que no quieren cumplir. Son libres de renegociar el contrato: “No puedo proporcionarle eso, pero, si me da esto, entonces podría proporcionarle otra cosa”.

Está claro que cualquier sistema que dependa de tecnología de agentes tiene una dependencia crítica de los acuerdos contractuales, incluso aunque se generen de manera dinámica.

Imagine: con los suficientes componentes y agentes que puedan negociar sus propios contratos entre ellos para conseguir un objetivo, podríamos resolver la crisis de productividad del software dejando que el software la resolviese por nosotros.

Pero, si no podemos usar contratos a mano, no seremos capaces de usarlos de manera automática. Así pues, la próxima vez que diseñe software, diseñe también su contrato.

Las secciones relacionadas incluyen

- Tema 24, “Los programas muertos no mienten”.
- Tema 25, “Programación asertiva”.
- Tema 38, “Programar por casualidad”.
- Tema 42, “Pruebas basadas en propiedades”.
- Tema 43, “Tenga cuidado ahí fuera”.
- Tema 45, “El pozo de los requisitos”.

Retos

- Puntos para reflexionar: si el DBC es tan potente, ¿por qué su uso no está más extendido? ¿Es difícil idear el contrato? ¿Nos hace pensar en problemas que preferiríamos ignorar por el momento? ¿¡Nos obliga a PENSAR!? ¡Está claro que es una herramienta peligrosa!

Ejercicios

Ejercicio 14

Diseñe una interfaz para una licuadora. Con el tiempo se convertirá en una licuadora basada en web y habilitada para el IoT, pero, por ahora, solo necesitamos la interfaz para controlarla. Tiene diez parámetros para la velocidad (0 significa apagada). No puede hacerla funcionar vacía y solo

puede cambiar la velocidad de unidad en unidad (es decir, de 0 a 1 y de 1 a 2, no de 0 a 2).

Estos son los métodos. Añada precondiciones y postcondiciones y una invariante.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void empty()
```

Ejercicio 15

¿Cuántos números hay en la serie 0, 5, 10, 15, ..., 100?

24 Los programas muertos no mienten

¿Se ha dado cuenta de que a veces otras personas pueden detectar que le pasa algo antes incluso de que usted mismo sea consciente del problema? Ocurre lo mismo con el código de otras personas. Si hay algo que está empezando a torcerse con uno de sus programas, a veces es una rutina de *framework* o biblioteca la que primero se da cuenta. Quizá ha pasado un valor `nil` o una lista vacía. Tal vez falta una clave en ese *hash*, o el valor que pensaba que contenía un *hash* en realidad contiene una lista en su lugar. Puede que haya un error de red o del sistema de archivos del que no se ha dado cuenta y ahora tiene datos vacíos o corruptos. Un error lógico un par de millones de instrucciones antes significa que el selector para una sentencia *case* ya no es el esperado 1, 2 o 3. Llegamos al caso predeterminado `default` de forma inesperada. Esa es también una razón por la que cada sentencia *case/switch* necesita tener una cláusula predeterminada; queremos saber cuándo ha pasado lo “imposible”.

Es fácil adoptar la mentalidad “no puede ocurrir”. La mayoría de nosotros hemos escrito código que no comprobaba si un archivo se cerraba con éxito, o si una sentencia *trace* se escribía como esperábamos. Y, en igualdad de condiciones, es probable que no lo necesitásemos; el código en cuestión no fallaría en condiciones normales. Pero estamos escribiendo código a la defensiva. Estamos asegurándonos de que los datos son los que

pensamos que son, que el código en producción es el código que pensamos que es. Estamos comprobando que las versiones correctas de las dependencias están cargadas de verdad.

Todos los errores proporcionan información. Podría convencerse de que no puede producirse un error y elegir ignorarlo, pero los programadores pragmáticos se dicen a sí mismos que, si hay un error, ha pasado algo muy, muy malo. No olvide leer el puñetero mensaje de error (consulte "Programador en tierras extrañas", en el tema 20).

La captura y liberación es para la pesca

Algunos desarrolladores consideran que un buen estilo es capturar o rescatar todas las excepciones y relanzarlas después de escribir algún tipo de mensaje. Su código está lleno de cosas como esta (donde una simple sentencia `raise` relanza la excepción actual):

```
try do
   add_score_to_board(score);
rescue InvalidScore
   Logger.error("Can't add invalid score. Exiting");
   raise
rescue BoardServerDown
   Logger.error("Can't add score: board is down. Exiting");
   raise
rescue StaleTransaction
   Logger.error("Can't add score: stale transaction. Exiting");
   raise
end
```

Los programadores pragmáticos escribirían esto así:

```
add_score_to_board(score);
```

Lo preferimos por dos razones. En primer lugar, el código de la aplicación no se ve eclipsado por el manejo de errores. En segundo lugar, y quizás más importante, el código está menos acoplado. En el ejemplo verboso, tenemos que incluir en una lista todas las excepciones que podría lanzar el método `add_score_to_board`. Si el escritor de ese método añade otra excepción, nuestro código se queda sutilmente desfasado. En la

segunda versión, más pragmática, la nueva excepción se propaga de forma automática.

Truco 38. Que el fallo total sea pronto.

Fallo total, no basura

Uno de los beneficios de detectar los problemas lo antes posible es que el fallo total puede producirse antes, y a menudo un fallo total es lo mejor que podemos hacer. La alternativa sería continuar, escribir datos corrompidos en alguna base de datos o mandar a la lavadora que realice su vigésimo ciclo de centrifugado consecutivo. Los lenguajes Erlang y Elixir abrazan esta filosofía. A Joe Armstrong, inventor de Erlang y autor de *Programming Erlang: Software for a Concurrent World* [Arm07], se le atribuyen a menudo las palabras: “La programación defensiva es una pérdida de tiempo. ¡Deja que falle!”. En estos entornos, los programas se diseñan para fallar, pero ese fallo se gestiona mediante supervisores. Un supervisor es responsable de ejecutar el código y sabe qué hacer si el código falla, lo cual podría incluir la limpieza, el reinicio, etc. ¿Qué pasa cuando es el supervisor en sí el que falla? Su propio supervisor gestiona ese evento, lo que lleva a un diseño compuesto por árboles de supervisores. La técnica es muy efectiva y ayuda a explicar el uso de estos lenguajes en sistemas de alta disponibilidad y con tolerancia a fallos.

En otros entornos, podría ser no ser apropiado salir sin más de un programa en ejecución. Puede que haya reclamado recursos que podrían no liberarse, o tal vez tenga que escribir mensajes de registro, organizar transacciones abiertas o interactuar con otros procesos. Sin embargo, el principio básico sigue siendo el mismo; cuando el código descubre que algo que se suponía que era imposible ha ocurrido, el programa ya no es viable. Cualquier cosa que haga a partir de ese momento pasa a ser sospechosa, así que acabe con él lo antes posible. En general, un programa muerto causa mucho menos daño que uno lisiado.

Las secciones relacionadas incluyen

- Tema 20, “Depuración”.
- Tema 23, “Diseño por contrato”.
- Tema 25, “Programación asertiva”.
- Tema 26, “Cómo equilibrar los recursos”.
- Tema 43, “Tenga cuidado ahí fuera”.

25 Programación asertiva

> *Hay cierto lujo en reprocharse algo a uno mismo. Cuando nos culpamos a nosotros mismos, sentimos que nadie más tiene derecho a culparnos.*
>
> *—Oscar Wilde, El retrato de Dorian Gray.*

Parece que hay un mantra que todo programador debe memorizar al principio de su carrera. Es un dogma fundamental de la informática, una creencia fundamental que aprendemos a aplicar a los requisitos, los diseños, el código, los comentarios, casi todo lo que hacemos. Dice:

> *Esto nunca puede pasar...*

“Esta aplicación nunca se usará en el extranjero, así que ¿por qué internacionalizarla?”, “`count` no puede ser negativo”, “el inicio de sesión no puede fallar”.

No practiquemos esta clase de autoengaño, sobre todo cuando escribamos código.

Truco 39. Use aserciones para evitar lo imposible.

Cada vez que se descubra pensando: “Pero, por supuesto, eso nunca podría ocurrir”, añada código para comprobarlo. La manera más fácil de hacerlo es utilizar aserciones. En muchas implementaciones de lenguajes, encontrará alguna forma de `assert` que comprueba una condición booleana.[2] Estas comprobaciones pueden ser muy valiosas. Si un parámetro o un resultado nunca debería ser `null`, compruébelo de manera explícita:

```
assert (result != null);
```

En la implementación de Java, puede (y debería) añadir una cadena descriptiva:

```
assert result != null && result.size() > 0 : “Empty result from XYZ”;
```

Las aserciones también son comprobaciones útiles de cómo opera un algoritmo. Quizá haya escrito un algoritmo de ordenamiento inteligente, llamado `my_sort`. Compruebe que funciona:

```
books = my_sort(find(“scifi”))
assert(is_sorted?(books))
```

No utilice aserciones en vez de manejo de errores real. Las aserciones comprueban si hay cosas que nunca deberían ocurrir: no le conviene escribir código como el siguiente:

```
puts(“Enter ‘Y’ or ‘N’: “)
ans = gets[0] # Toma el primer carácter de respuesta
assert((ch == ‘Y’) || (ch == ‘N’)) # ¡Muy mala idea!
```

Y solo porque la mayoría de las implementaciones de `assert` pondrán fin al proceso cuando una aserción falle, no hay razón para que las versiones que escriba usted también deberían. Si necesita liberar recursos, capture la excepción de la aserción o atrape la salida, y ejecute su propio manejador de errores. Asegúrese de que el código que ejecuta en esos milisegundos agonizantes no depende de la información que desencadenó el fallo de la aserción en primer lugar.

Aserciones y efectos secundarios

Es embarazoso que el código que añadimos para detectar errores acabe en realidad creando errores nuevos. Esto puede ocurrir con las aserciones si evaluar la condición tiene efectos secundarios. Por ejemplo, sería una mala idea escribir código para algo como:

```
while (iter.hasMoreElements()) {
   assert(iter.nextElement() != null);
   Object obj = iter.nextElement();
   // ....
}
```

La llamada a `.nextElement()` en la aserción tiene el efecto secundario de hacer que el iterador pase de largo del elemento que se está cogiendo, y, por tanto, el bucle procesará solo la mitad de los elementos en la colección. Sería mejor escribir:

```
while (iter.hasMoreElements()) {
   Object obj = iter.nextElement();
   assert(obj != null);
   // ....
}
```

Este problema es un tipo de Heisenbug, una depuración que cambia el comportamiento del sistema que se está depurando.

(También creemos que hoy en día, cuando la mayoría de los lenguajes tienen un soporte decente para iterar funciones sobre colecciones, este tipo de bucle explícito es innecesario y de mala forma).

Deje las aserciones activadas

Existe un malentendido común acerca de las aserciones. Es algo así:

> *Las aserciones añaden cierta tara al código. Puesto que comprueban si hay cosas que nunca deberían pasar, se desencadenan solo por un fallo en el código. Una vez que el código se ha probado y enviado, ya no son necesarias, y deberían desactivarse para hacer que el código se ejecute más rápido. Las aserciones son una herramienta de depuración.*

Aquí hay dos supuestos evidentemente erróneos. En primer lugar, se asume que las pruebas encuentran todos los fallos. En realidad, para cualquier programa complejo, es poco probable que pruebe siquiera un minúsculo porcentaje de las permutaciones a las que se verá sometido el código. En segundo lugar, los optimistas están olvidando que el programa se ejecuta en un mundo peligroso. Durante las pruebas, no es probable que las ratas roan un cable de comunicaciones, nadie agotará la memoria jugando a un juego y los archivos de registro no llenarán la partición de almacenamiento. Estas cosas podrían pasar cuando el programa se ejecute en un entorno de producción. La primera línea de defensa es buscar cualquier posible error y la segunda es utilizar aserciones para intentar detectar aquellos que hemos pasado por alto.

Desactivar las aserciones cuando se entrega el programa a producción es como cruzar un alambre a gran altura sin red porque una vez lo hicimos al practicar. Hay un valor dramático, pero es difícil conseguir un seguro de vida.

Incluso si tiene problemas de rendimiento, desactive solo aquellas aserciones que realmente le afecten. El ejemplo de ordenamiento que hemos visto antes podría ser una parte crítica de su aplicación y podría ser necesario que fuese rápido. Añadir la comprobación significa otra pasada por los datos, lo cual podría ser inaceptable. Haga que esa comprobación en particular sea opcional, pero deje el resto en su sitio.

Use aserciones en producción y gane un dineral

Un antiguo vecino de Andy estaba al frente de una pequeña *startup* que fabricaba dispositivos de red. Uno de los secretos de su éxito fue la decisión de dejar las aserciones en su sitio en las liberaciones a producción. Esas aserciones estaban bien elaboradas para informar de todos los datos pertinentes que llevaban al fallo y se presentaban mediante una bonita interfaz de usuario al usuario final. Este nivel de *feedback*, de usuarios reales en condiciones auténticas, permitía a los desarrolladores tapar los agujeros y arreglar aquellos fallos oscuros y difíciles de reproducir, lo que tenía como resultado un software muy estable y a prueba de bombas.
Esta empresa pequeña y desconocida tenía un producto tan sólido que pronto se vendió por cientos de millones de dólares.
Así, a modo de anécdota.

Ejercicios

Ejercicio 16

Un bofetón de realidad. ¿Cuál de estas cosas “imposibles” puede ocurrir?

- Un mes con menos de 28 días.
- Código de error de una llamada al sistema: no se puede acceder al directorio actual.
- En C++: a = 2; b = 3; pero (a + b) no es igual a 5.
- Un triángulo con una suma de ángulos interiores ≠ 180°.
- Un minuto que no tiene 60 segundos.

- (a + 1) <= a.

Las secciones relacionadas incluyen

- Tema 23, “Diseño por contrato”.
- Tema 24, “Los programas muertos no mienten”.
- Tema 42, “Pruebas basadas en propiedades”.
- Tema 43, “Tenga cuidado ahí fuera”.

26 Cómo equilibrar los recursos

Cuando enciendes una vela, también creas una sombra...

—Ursula K. Le Guin, Un mago de Terramar.

Todos gestionamos recursos cada vez que escribimos código: memoria, transacciones, hilos, conexiones de red, archivos, temporizadores, todo tipo de cosas con disponibilidad limitada. La mayor parte del tiempo, el uso de recursos sigue un patrón predecible: se asigna el recurso, se utiliza y se desasigna.

Sin embargo, muchos desarrolladores no tienen un plan consistente para tratar la asignación y desasignación de recursos, así que vamos a sugerir un truco sencillo:

Truco 40. Acabe lo que empiece.

Este truco es fácil de aplicar en la mayoría de las circunstancias. Solo significa que la función o el objeto que asigna un recurso debería ser responsable de desasignarlo. Vamos a ver cómo se aplica fijándonos en un ejemplo de código malo; es parte de un programa de Ruby que abre un archivo, lee en él información del cliente, actualiza un campo y escribe el resultado de vuelta. Hemos eliminado el manejo de errores para hacer el ejemplo más claro:

```
def read_customer
  @customer_file = File.open(@name + ".rec", "r+")
  @balance = BigDecimal(@customer_file.gets)
```

```
end

def write_customer
   @customer_file.rewind
   @customer_file.puts @balance.to_s
   @customer_file.close
end

def update_customer(transaction_amount)
   read_customer
   @balance = @balance.add(transaction_amount,2)
   write_customer
end
```

A primera vista, la rutina `update_customer` parece razonable. Parece implementar la lógica que requerimos: leer un registro, actualizar el balance y volver a escribir el registro de nuevo. Sin embargo, esta pulcritud oculta un problema importante. Las rutinas `read_customer` y `write_customer` tienen un acoplamiento fuerte;[3] comparten la variable de instancia `customer_file`. `read_customer` abre el archivo y almacena la referencia del archivo en `customer_file`, y, después, `write_customer` usa esa referencia almacenada para cerrar el archivo cuando acaba. Esta variable compartida ni siquiera aparece en la rutina `update_customer`.

¿Por qué es malo esto? Pensemos en el desgraciado programador encargado del mantenimiento al que se le dice que la especificación ha cambiado; el balance debería actualizarse solo si el nuevo valor no es negativo. El programador va a la fuente y cambia `update_customer`:

```
def update_customer(transaction_amount)
   read_customer
   if (transaction_amount >= 0.00)
      @balance = @balance.add(transaction_amount,2)
      write_customer
   end
end
```

Todo parece ir bien durante las pruebas. Sin embargo, cuando el código pasa a producción, se viene abajo después de unas horas, quejándose de que hay demasiados archivos abiertos. Resulta que no se está llamando a

`write_customer` en algunas circunstancias. Cuando eso ocurre, el archivo no está cerrándose.

Una solución muy mala a este problema sería tratar con el caso especial en `update_customer:`.

```
def update_customer(transaction_amount)
   read_customer
   if (transaction_amount >= 0.00)
      @balance += BigDecimal(transaction_amount, 2)
      write_customer
   else
      @customer_file.close # ¡Mala idea!
   end
end
```

Esto arreglará el problema; ahora, el archivo se cerrará al margen del nuevo balance, pero este arreglo significa ahora que hay tres rutinas acopladas a través de la variable compartida `customer_file`, y hacer un seguimiento de cuándo se abre o no el archivo va a empezar a crear un gran lío. Estamos cayendo en una trampa y las cosas van a empezar a ir cuesta abajo enseguida si seguimos por este camino. ¡Esto no está equilibrado!

El truco “Acabe lo que empiece” nos dice que, de manera ideal, la rutina que asigna un recurso también debería liberarlo. Podemos aplicarlo aquí haciendo una pequeña refactorización del código:

```
def read_customer(file)
   @balance=BigDecimal(file.gets)
end

def write_customer(file)
   file.rewind
   file.puts @balance.to_s
end

def update_customer(transaction_amount)
   file=File.open(@name + ".rec", "r+") # >—
   read_customer(file) # |
   @balance = @balance.add(transaction_amount,2) # |
   write_customer(file) # |
   file.close # <—
end
```

En vez de aferrarnos a la referencia del archivo, hemos cambiado el código para pasarla como un parámetro.[4] Ahora toda la responsabilidad sobre el archivo está en la rutina `update_customer`. Abre el archivo y (acabando lo que empieza) lo cierra antes de volver. La rutina equilibra el uso del archivo: la apertura y el cierre están en el mismo sitio y resulta evidente que para cada apertura habrá un cierre correspondiente. La refactorización también elimina una variable compartida muy fea.

Hay otra mejora pequeña, pero importante, que puede hacer. En muchos lenguajes modernos, se puede delimitar el tiempo de vida de un recurso a un bloque cerrado de algún tipo. En Ruby, hay una variación del archivo `open` que pasa la referencia del archivo abierto a un bloque, que aquí se muestra entre el `do` y el `end`:

```
def update_customer(transaction_amount)
   File.open(@name + ".rec", "r+") do |file| # >—
      read_customer(file) # |
      @balance = @balance.add(transaction_amount,2) # |
      write_customer(file) # |
   end # <—
end
```

En este caso, al final del bloque la variable `file` se sale del ámbito y el archivo externo se cierra. Punto. No hace falta acordarse de cerrar el archivo y liberar la fuente, está garantizado que se hará por usted.

En caso de duda, siempre compensa reducir el alcance.

Truco 41. Actúe a nivel local.

Anidar asignaciones

El patrón básico para la asignación de recursos puede ampliarse para rutinas que necesitan más de un recurso a la vez. Aquí hay dos sugerencias más:

- Desasigne recursos en el orden inverso al que utilizó para asignarlos. De ese modo, no dejará huérfano a ningún recurso si uno de los recursos contiene referencias a otro.

- Cuando asigne el mismo conjunto de recursos en diferentes puntos de su código, asígnelos siempre en el mismo orden. Esto reducirá las posibilidades de que se produzca un bloqueo mutuo. (Si el proceso A reclama `resource1` y está a punto de reclamar `resource2`, mientras que el proceso B ha reclamado `resource2` y está intentando hacerse con `resource1`, los dos procesos se quedarán esperando para siempre).

No importa qué tipo de recursos estemos utilizando (transacciones, conexiones de red, memoria, archivos, hilos, ventanas), se aplica el patrón básico: quienquiera que asigne un recurso debería ser responsable de desasignarlo. Sin embargo, en algunos lenguajes podemos desarrollar más el concepto.

Equilibrar a lo largo del tiempo

En este tema estamos fijándonos sobre todo en recursos efímeros utilizados por el proceso en ejecución, pero puede que le convenga pensar en otros líos que podría estar dejando a su paso.
Por ejemplo, ¿cómo se gestionan sus archivos de registro? Está creando datos y utilizando espacio de almacenamiento. ¿Cuenta con algo para rotar los registros y limpiarlos? ¿Y para los archivos de depuración no oficiales que está soltando? Si está añadiendo registros a una base de datos, ¿cuenta con algún proceso similar para que expiren? Para cualquier cosa que cree que ocupe un recurso finito, plantéese la manera de equilibrarla.
¿Qué más está dejando atrás?

Objetos y excepciones

El equilibrio entre asignaciones y desasignaciones recuerda al constructor y el destructor de una clase orientada a objetos. La clase representa un recurso, el constructor nos da un objeto particular de ese tipo de recurso y el destructor lo elimina de nuestro ámbito.

Si está programando en un lenguaje orientado a objetos, puede que le resulte útil encapsular recursos en clases. Cada vez que necesite un tipo de recurso particular, instanciará un objeto de esa clase. Cuando el objeto se

salga del ámbito o lo reclame el recolector de basura, el destructor del objeto desasignará el recurso envuelto.

Este enfoque tiene beneficios concretos cuando se trabaja con lenguajes donde las excepciones pueden interferir con la desasignación de recursos.

Equilibrio y excepciones

Los lenguajes que soportan excepciones pueden complicar la desasignación de recursos. Si se lanza una excepción, ¿cómo garantiza que todo lo asignado antes de la excepción está ordenado? La respuesta depende hasta cierto punto del soporte del lenguaje. Por lo general, tiene dos opciones:

1. Usar el ámbito de la variable (por ejemplo, apilar variables en C++ o Rust).
2. Usar una cláusula `finally` en un bloque `try...catch`.

Con las reglas habituales sobre ámbitos en lenguajes como C++ o Rust, se reclamará la memoria de la variable cuando esta salga del ámbito mediante un retorno, salida de bloque o excepción. Pero también puede conectarse al destructor de la variable para limpiar cualquier recurso externo. En este ejemplo, la variable de Rust llamada `accounts` cerrará de manera automática el archivo asociado cuando se salga del ámbito:

```
{
let mut accounts = File::open(“mydata.txt”)?; // >—
// usa ‘accounts’ // |
... // |
} // <—
// ‘accounts’ está ahora fuera del ámbito y el archivo se
// cierra automáticamente
```

La otra opción, si el lenguaje la soporta, es la cláusula `finally`. Una cláusula `finally` se asegurará de que el código especificado se ejecute tanto si se ha lanzado una excepción como si no en el bloque `try...catch`:

```
try
  // alguna cosa sospechosa
```

```
catch
   // se lanzó una excepción
finally
   // limpiar en cualquier caso
```

Sin embargo, hay trampa.

Un antipatrón de excepciones

Con frecuencia, vemos a gente que escribe algo así:

```
begin
   thing = allocate_resource()
   process(thing)
finally
   deallocate(thing)
end
```

¿Ve dónde está el problema?

¿Qué pasa si la asignación de recursos falla y lanza una excepción? La cláusula `finally` la capturará e intentará desasignar una cosa que nunca se ha asignado.

El patrón correcto para manejar la desasignación de recursos en un entorno con excepciones es:

```
thing = allocate_resource()
begin
   process(thing)
finally
   deallocate(thing)
end
```

Cuando no podemos equilibrar los recursos

Hay veces en las que el patrón de asignación de recursos básico no es apropiado. Por lo general, eso ocurre con programas que utilizan estructuras de datos dinámicas. Una rutina asignará un área de memoria y la vinculará a una estructura mayor, donde puede que permanezca un tiempo.

Aquí el truco está en establecer una invariante semántica para la asignación de memoria. Necesita decidir quién es responsable de los datos

en una estructura de datos agregada. ¿Qué pasa cuando se desasigna la estructura del nivel superior? Tiene tres opciones principales:

- La estructura del nivel superior también es responsable de liberar cualquier subestructura que contiene. Después, estas estructuras pueden eliminar de forma recursiva datos que contienen, y así sucesivamente.
- La estructura del nivel superior se desasigna simplemente. Cualquier estructura a la que apuntase (y no esté referenciada en ninguna otra parte) se queda huérfana.
- La estructura del nivel superior se niega a desasignarse a sí misma si contiene subestructuras.

Aquí la elección depende de las circunstancias de cada estructura de datos individual. Sin embargo, necesita hacer que sea explícito para cada una de ellas e implementar su decisión de manera coherente. Implementar cualquiera de estas acciones en un lenguaje por procedimientos como C puede ser un problema: las estructuras de datos en sí no están activas. En estas circunstancias, nosotros preferimos escribir un módulo para cada estructura importante que proporcione servicios de asignación y desasignación estándar para esa estructura. (Este módulo también puede proporcionar servicios como la impresión de depuración, la serialización, la deserialización y enganches de recorrido).

Comprobar el equilibrio

Puesto que los programadores pragmáticos no confiamos en nadie, ni siquiera en nosotros mismos, nos parece que siempre es buena idea construir código que compruebe de verdad que los recursos se liberan de manera apropiada. Para la mayoría de las aplicaciones, esto suele significar producir envoltorios para cada tipo de recurso y utilizar estos envoltorios para hacer un seguimiento de todas las asignaciones y desasignaciones. En determinados puntos del código, la lógica del programa dictará que los recursos estarán en un estado determinado: utilice los envoltorios para comprobar esto. Por ejemplo, un programa de larga duración que atienda solicitudes tendrá probablemente un solo punto en la parte superior de su

bucle de procesamiento principal donde espera a que llegue la siguiente solicitud. Este es un buen lugar para garantizar que el uso de los recursos no ha aumentado desde la última ejecución del bucle.

A un nivel inferior, pero no menos útil, puede invertir en herramientas que (entre otras cosas) comprueban sus programas en ejecución para ver si hay fugas de memoria.

Las secciones relacionadas incluyen

- Tema 24, “Los programas muertos no mienten”.
- Tema 30, “Transformar la programación”.
- Tema 33, “Romper el acoplamiento temporal”.

Retos

- Aunque no hay maneras garantizadas de asegurar que siempre liberamos los recursos, algunas técnicas de diseño, cuando se aplican de forma coherente, ayudan. En el texto hemos hablado de cómo una invariante semántica para estructuras de datos importantes podría dirigir las decisiones de desasignación de memoria. Plantéese cómo el tema 23, “Diseño por contrato”, podría ayudar a perfeccionar esta idea.

Ejercicios

Ejercicio 17

Algunos desarrolladores de C y C++ insisten en establecer un puntero en `NULL` después de desasignar la memoria a la que hace referencia. ¿Por qué es buena idea?

Ejercicio 18

Algunos desarrolladores de Java insisten en establecer una variable de objeto en `NULL` después de haber terminado de utilizar el objeto. ¿Por qué es buena idea?

27 No vaya más rápido que sus faros

Es difícil hacer predicciones, sobre todo acerca del futuro.

—Lawrence "Yogi" Berra, a partir de un proverbio danés.

Es de noche, está oscuro y llueve a mares. El descapotable derrapa por las curvas cerradas pequeñas y sinuosas de la montaña, apenas aguantando los giros. Aparece una curva muy cerrada y el coche no puede trazarla, choca contra el precario guardarraíl y vuela hasta estrellarse en el valle de abajo. La policía estatal llega a la escena y el agente al mando niega con la cabeza, apesadumbrado. "Debe haber ido más rápido que sus faros".

¿Iba el descapotable lanzado más rápido que la velocidad de la luz? No, ese límite de velocidad está bien fijado. A lo que se refiere el agente es a la capacidad del conductor para detenerse o girar el volante a tiempo en respuesta a la iluminación de los faros.

Los faros tienen un alcance limitado, conocido como distancia de proyección. Pasado ese punto, la dispersión de la luz es demasiado difusa para ser efectiva. Además, los faros solo proyectan en línea recta, y no iluminarán nada fuera del eje, como curvas, colinas o baches en la carretera. Según la Administración Nacional de Seguridad del Tráfico en las Carreteras de EE. UU. (NHTSA), la distancia media iluminada por faros de corto alcance es de algo menos de 50 metros. Por desgracia, la distancia de detención a 64,3 km/h es de 57,6 metros y a 112,6 km/h, la barbaridad de 141,4 metros.[5] Así pues, lo cierto es que es bastante fácil ir más rápido que nuestros faros. En el desarrollo de software, nuestros "faros" están limitados de manera similar. No podemos ver mucho más allá en el futuro y, cuanto más nos desviamos del eje, más oscuro está todo. Por tanto, los programadores pragmáticos tienen una norma firme:

Truco 42. Avance a pasos pequeños. Siempre.

Dé siempre pasos pequeños y deliberados, busque *feedback* y realice ajustes antes de continuar. Piense que la tasa de *feedback* es su límite de velocidad. Nunca dé un paso o emprenda una tarea que sea "demasiado grande".

¿A qué nos referimos exactamente con *feedback*? Cualquier cosa que confirme o desmienta de forma independiente su acción. Por ejemplo:

- Los resultados de un REPL proporcionan *feedback* acerca de su comprensión de las API y los algoritmos.
- Las pruebas unitarias proporcionan *feedback* sobre su último cambio en el código.
- Las demostraciones a usuarios y las conversaciones proporcionan *feedback* sobre las características y la usabilidad.

¿Qué es una tarea demasiado grande? Cualquiera que requiera "adivinación". Al igual que los faros del coche tienen un alcance limitado, nosotros solo podemos anticipar uno o dos pasos en el futuro, quizá algunas horas o unos días como mucho. Más allá de eso, pasamos de una suposición fundada a una especulación salvaje. Puede que se encuentre en el terreno de la adivinación cuando tenga que:

- Estimar fechas de completitud para dentro de unos meses.
- Planear un diseño para el mantenimiento o la ampliación en el futuro.
- Adivinar las necesidades futuras del usuario.
- Adivinar la disponibilidad tecnológica del futuro.

Sí, le estamos oyendo gritar, ¿no se supone que tenemos que diseñar pensando en el futuro mantenimiento? Sí, pero solo hasta cierto punto: solo hasta donde podamos ver. Cuanto más tengamos que predecir cómo será el futuro, mayor riesgo corremos de equivocarnos. En vez de malgastar esfuerzos en diseñar pensando en un futuro incierto, siempre podemos recurrir a diseñar nuestro código para que sea reemplazable. Haga que sea fácil desechar su código y sustituirlo por algo más adecuado. Hacer que el código sea reemplazable también ayudará con la cohesión, el acoplamiento y el principio DRY, lo que llevará a un mejor diseño general.

Incluso aunque se sienta seguro respecto al futuro, siempre existe la posibilidad de que haya un cisne negro a la vuelta de la esquina.

Cisnes negros

En su libro *El cisne negro: El impacto de lo altamente improbable* [Tal10], Nassim Nicholas Taleb sugiere que todos los acontecimientos significativos de la historia han derivado de eventos de perfil alto, infrecuentes y difíciles de predecir que están más allá del reino de las expectativas normales. Estos casos anómalos, aunque son poco frecuentes a nivel estadístico, tienen efectos desproporcionados. Además, nuestro propio sesgo cognitivo tiende a cegarnos ante los cambios que acechan en los límites de nuestro trabajo (consulte “Sopa de piedras y ranas hervidas”).

Más o menos en la misma época que la primera edición de *El programador pragmático*, había un debate candente en los foros en línea y las revistas de informática acerca de una pregunta polémica: “¿Quién ganaría la guerra de las GUI para escritorio, Motif u OpenLook?”.[6] Era la pregunta equivocada. Lo más probable es que nunca haya oído hablar de estas tecnologías, ya que ninguna “ganó” y la web centrada en navegadores enseguida dominó el panorama.

Truco 43. Evite la adivinación.

La mayor parte del tiempo, el mañana se parece mucho al hoy, pero no lo dé por hecho.

Las secciones relacionadas incluyen

- Tema 12, “Balas trazadoras”.
- Tema 13, “Prototipos y notas en *post-its*”.
- Tema 40, “Refactorización”.
- Tema 41, “Probar para escribir código”.
- Tema 48, “La esencia de la agilidad”.
- Tema 50, “Los cocos no sirven”.

---

[1] Basado en parte en trabajos anteriores de Dijkstra, Floyd, Hoare, Wirth y otros.

[2] En C y C++ esto suele implementarse como macros. En Java, las aserciones están deshabilitadas por defecto. Invoque la máquina virtual Java VM con la bandera `–enableassertions` para

habilitarlas y déjelas habilitadas.

[3] Para ver una explicación sobre los peligros del código acoplado, consulte el tema 28, "Desacoplamiento".

[4] Consulte el truco 50.

[5] Según la NHTSA, la distancia de detención = distancia de reacción + distancia de frenado, asumiendo que hay un tiempo medio de reacción de 1,5 s y deceleración de 5,15 m/s^2.

[6] Motif y OpenLook eran estándares de GUI para X-Windows basados en estaciones de trabajo de Unix.

5
O se adapta o se rompe

La vida no se queda quieta, y tampoco puede hacerlo el código que escribimos. Para poder mantenernos al día con el ritmo casi frenético del cambio, necesitamos esforzarnos por escribir código que sea lo más flexible posible. De lo contrario, nos encontraremos con un código que enseguida se queda obsoleto, o es demasiado frágil para arreglarlo, y que al final se queda atrás en la loca carrera hacia el futuro.

En “Reversibilidad” hablábamos de los peligros de las decisiones irreversibles. En este capítulo, explicaremos cómo tomar decisiones reversibles para que el código pueda mantenerse flexible y adaptable en un mundo incierto.

Primero, hablaremos del acoplamiento, las dependencias entre porciones de código. “Desacoplamiento” muestra cómo mantener separados los conceptos que van separados, reduciendo el acoplamiento.

Después, echaremos un vistazo a diferentes técnicas que puede utilizar al hacer “Malabares con el mundo real”. Examinaremos cuatro estrategias diferentes para ayudar a gestionar y reaccionar a eventos, un aspecto crítico de las aplicaciones de software modernas.

El código orientado a objetos y por procedimientos tradicional podría tener un acoplamiento demasiado fuerte para nuestros fines. En “Transformar la programación”, aprovecharemos las ventajas del estilo más flexible y más claro ofrecido por las *pipelines* de funciones, incluso si nuestro lenguaje no las soporta de manera directa.

El estilo orientado a objetos habitual puede atraerle hacia otra trampa. No caiga en ella o acabará pagando un “Impuesto sobre la herencia” considerable. Exploraremos alternativas mejores para mantener el código flexible y más fácil de cambiar.

Y, por supuesto, una buena manera de mantener la flexibilidad es escribir menos código. Cambiar el código abre la posibilidad de que se introduzcan

fallos nuevos. “Configuración” explicará cómo extraer detalles del código por completo, donde puedan modificarse de manera más segura y sencilla.

Todas estas técnicas le ayudarán a escribir código que se adapte y no se rompa.

28 Desacoplamiento

Cuando tratamos de elegir algo por sí mismo, lo encontramos atado a todo lo demás en el universo.

—John Muir, Mi primer verano en la Sierra.

En el tema 8, “La esencia del buen diseño”, afirmamos que utilizar buenos principios de diseño hará que el código que escribimos sea fácil de cambiar. El acoplamiento es el enemigo del cambio, porque vincula cosas que deben cambiar en paralelo. Esto hace que el cambio sea más difícil: dedicamos tiempo a rastrear todas las partes que necesitan cambiarse o lo dedicamos a preguntarnos por qué las cosas se estropearon cuando modificamos “solo una cosa” y no las demás con las que estaba acoplada.

Cuando diseñamos algo que queremos que sea rígido, como un puente o una torre, quizá, acoplamos los componentes (figura 5.1):

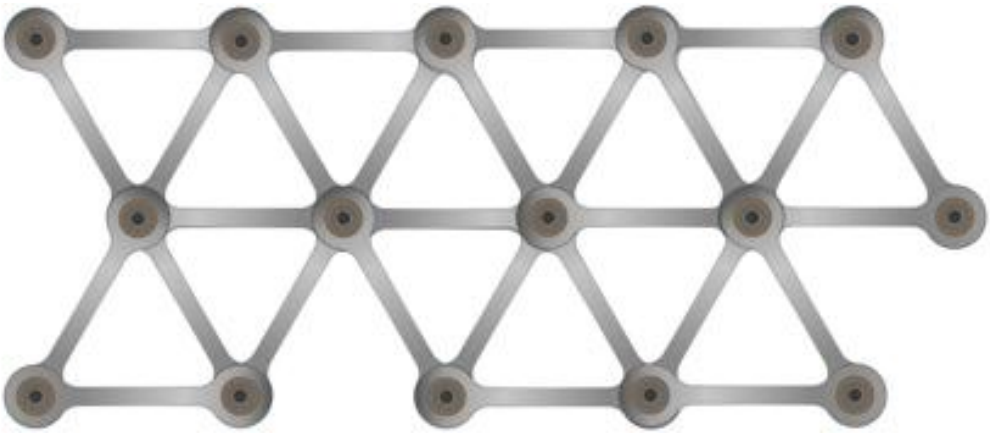


Figura 5.1.

Las piezas trabajan juntas para hacer que la estructura sea rígida.

Compare eso con algo como esto (figura 5.2):

Figura 5.2.

Aquí no hay rigidez estructural: las piezas individuales pueden cambiar y otras solo se acomodan.

Si diseñamos puentes, queremos que mantengan su forma; necesitamos que sean rígidos. Pero, si diseñamos software que querremos modificar, queremos justo lo contrario: queremos que sea flexible. Y, para que sea flexible, los componentes individuales deberían acoplarse con la menor cantidad posible de otros componentes.

Para empeorar las cosas, el acoplamiento es transitivo: si A está acoplado con B y C, y B está acoplado con M y N, y C con X e Y, entonces en realidad A está acoplado con B, C, M, N, X e Y. Eso significa que hay un principio simple que debería seguir:

Truco 44. El código desacoplado es más fácil de cambiar.

Puesto que por lo general no escribimos código utilizando vigas de acero y remaches, ¿qué significa desacoplar código? En esta sección hablaremos de:

- ***Train wrecks* (choque de trenes):** Cadenas de llamadas a métodos.
- **Globalización:** Los peligros de las cosas estáticas.
- **Herencia:** Por qué la creación subclases es peligrosa.

Hasta cierto punto, esta lista es artificial: el acoplamiento puede producirse en cualquier momento en que dos porciones de código compartan algo, así que, mientras lea lo que se explica a continuación, preste atención a los patrones subyacentes para poder aplicarlos a su código. Y manténgase atento a cualquier señal de acoplamiento:

- Dependencias absurdas entre módulos o bibliotecas no relacionados.
- Cambios “simples” en un módulo que se propagan a través de módulos no relacionados en el sistema o estropean algo en cualquier otra parte de este.
- Desarrolladores que tienen miedo de modificar el código porque no están seguros de qué puede verse afectado.
- Reuniones a las que tiene que asistir todo el mundo porque nadie está seguro de a quién afectará el cambio.

Train Wrecks

Todos hemos visto (y probablemente escrito) código como este:

```
public void applyDiscount(customer, order_id, discount) {
   totals = customer
        .orders
        .find(order_id)
        .getTotals();
   totals.grandTotal = totals.grandTotal–discount;
   totals.discount = discount;
}
```

Estamos obteniendo una referencia a algunos pedidos (`orders`) desde un objeto cliente (`customer`), usando eso para encontrar un pedido en particular y, después, obteniendo el conjunto de totales (`totals`) para el pedido.

Utilizando esos totales, restamos el descuento de la suma total del pedido y también los actualizamos con ese descuento.

Esta porción de código está atravesando cinco niveles de abstracción, desde el cliente a las cantidades totales. En última instancia, nuestro código de nivel superior debe saber que un objeto cliente expone pedidos, que los pedidos tienen un método `find` que toma el identificador de un pedido y devuelve un pedido y que el objeto pedido tiene un objeto `totals` que tiene *getters* y *setters* para las sumas totales y los descuentos. Ahí hay un montón de conocimiento implícito, pero lo peor es que hay muchas cosas que no pueden cambiar en el futuro si se pretende que este código siga funcionado. Todos los vagones de un tren están acoplados, al igual que todos los métodos y atributos en un *train wreck*.

Supongamos que la empresa decide que ningún pedido puede tener un descuento de más del 40 %. ¿Dónde pondríamos el código que impone esa regla?

Podríamos decir que pertenece a la función `applyDiscount` que acabamos de escribir. Desde luego, esa es parte de la respuesta. Pero, con el código tal y como es ahora, no podemos saber que esta es la respuesta completa. Cualquier porción de código, en cualquier parte, podría establecer campos en el objeto `totals` y, si el responsable del mantenimiento de ese código no recibió la circular, no estaría comprobando si se cumple la nueva política.

Una forma de ver esto es pensar en las responsabilidades. Está claro que el objeto `totals` debería ser responsable de gestionar los totales. Y, aun así, no lo es: en realidad es solo un contenedor para un puñado de campos que cualquiera puede consultar y actualizar. La solución para eso es aplicar lo que llamamos:

Truco 45. *Tell, Don't Ask* (mande, no pregunte).

Este principio dice que no deberíamos tomar decisiones basadas en el estado interno de un objeto y después actualizar ese objeto. Si se hace eso, se destruyen por completo los beneficios del encapsulamiento y, al hacerlo, se extiende la información de la implementación por todo el código. Por tanto, lo primero que hay que hacer para arreglar nuestro *train wreck* es delegar la aplicación de descuentos al objeto total:

```
public void applyDiscount(customer, order_id, discount) {
   customer
      .orders
      .find(order_id)
      .getTotals()
      .applyDiscount(discount);
}
```

Tenemos el mismo tipo de problema TDA (*tell*, *don't ask*) con el objeto cliente y sus pedidos: no deberíamos tomar su lista de pedidos y buscar entre ellos, sino que deberíamos obtener el pedido que queremos directamente del cliente:

```
public void applyDiscount(customer, order_id, discount) {
   customer
      .findOrder(order_id)
      .getTotals()
      .applyDiscount(discount);
}
```

Lo mismo se aplica a nuestro objeto pedido y sus totales. ¿Por qué debería el mundo exterior tener que saber que la implementación de un pedido utiliza un objeto separado para almacenar sus totales?

```
public void applyDiscount(customer, order_id, discount) {
   customer
      .findOrder(order_id)
      .applyDiscount(discount);
}
```

Y aquí es donde probablemente pararíamos.

En este punto, puede que esté pensando que TDA nos haría añadir un método `applyDiscountToOrder(order_id)` a los clientes. Y, si se sigue al pie de la letra, sería así.

Pero TDA no es una ley de la naturaleza; es solo un patrón que nos ayuda a reconocer problemas. En este caso, estamos cómodos exponiendo el hecho de que un cliente tiene pedidos y que podemos encontrar uno de esos pedidos pidiéndoselo al objeto cliente. Esta es una decisión pragmática.

En toda aplicación hay determinados conceptos de nivel superior que son universales. En esta aplicación, esos conceptos incluyen clientes y pedidos. No tiene sentido ocultar por completo los pedidos dentro de los objetos cliente: tienen una existencia propia. Así pues, no tenemos problema con crear API que expongan objetos pedido.

La ley de Demeter

A menudo, la gente habla de algo denominado “ley de Demeter”, o LoD por sus siglas en inglés, en relación con el acoplamiento. La LoD es un conjunto de directrices[1] escritas a finales de los ochenta por Ian Holland. La creó para ayudar a los desarrolladores del proyecto Demeter a mantener sus funciones más limpias y desacopladas.

La LoD dice que un método definido en una clase C solo debería llamar a:

- Otros métodos de la instancia en C.
- Sus parámetros.
- Métodos en objetos que cree, tanto en la pila como en el montículo.
- Variables globales.

En la primera edición de este libro, dedicamos un tiempo a describir la LoD. En los 20 años transcurridos, el capullo de esa rosa concreta se ha marchitado un poco. Ahora no nos gusta la cláusula de la "variable global" (por razones que veremos en la siguiente sección). También hemos descubierto que es difícil utilizar esto en la práctica: es un poco parecido a tener que analizar sintácticamente un documento legal cada vez que se llama a un método.

Sin embargo, el principio sigue siendo sensato. Simplemente recomendamos una manera en cierto modo más simple de expresar casi lo mismo:

Truco 46. No encadene llamadas a métodos.

Intente no tener más de un "." cuando acceda a algo. Y "acceder a algo" también se aplica a casos en los que utiliza variables intermedias, como en el siguiente código:

```
# Este estilo es bastante pobre
amount = customer.orders.last().totals().amount;

# y este también...
orders = customer.orders;
last = orders.last();
totals = last.totals();
amount = totals.amount;
```

Hay una gran excepción a la regla de "un punto": la regla no se aplica si las cosas que estamos encadenando tienen muy muy pocas probabilidades de cambiar. En la práctica, se debería considerar que cualquier cosa en la aplicación tiene probabilidades de cambiar. Cualquier cosa que haya en una

biblioteca de terceros debería considerarse volátil, sobre todo si se sabe que los mantenedores de esa biblioteca cambian API entre liberaciones. Sin embargo, es probable que las bibliotecas que vienen con el lenguaje sean bastante estables, así que podríamos estar satisfechos con un código como:

```
people
.sort_by {|person| person.age }
.first(10)
.map {| person | person.name }
```

Ese código Ruby funcionaba cuando escribimos la primera edición, hace 20 años, y es probable que siga funcionando cuando nos vayamos al asilo para programadores (cualquier día de estos...).

Cadenas y *pipelines*

En el tema 30, “Transformar la programación”, hablamos de componer funciones en *pipelines*. Estas *pipelines* transforman datos, pasándolos de una función a la siguiente. Esto no es lo mismo que un *train wreck* de llamadas a métodos, puesto que no dependemos de detalles de implementación ocultos. Eso no quiere decir que las *pipelines* no introduzcan cierto acoplamiento: lo hacen. El formato de los datos devueltos por una función en una *pipeline* debe ser compatible con el formato aceptado por la siguiente.

Según nuestra experiencia, esta forma de acoplamiento es una barrera mucho más pequeña para cambiar el código que la forma introducida por los *train wrecks*.

Los males de la globalización

Los datos accesibles a nivel global son una fuente traicionera de acoplamiento entre componentes de la aplicación. Cada porción de los datos globales actúa como si cada método de la aplicación ganase de repente un parámetro adicional: al fin y al cabo, esos datos globales están disponibles dentro de cada método. Los datos globales acoplan código por muchas razones. La más obvia es que un cambio en la implementación de los datos globales afecta de manera potencial a todo el código del sistema. En la

práctica, por supuesto, el impacto es bastante limitado; en realidad, el problema se reduce a saber que hemos encontrado todos los lugares que necesitamos cambiar. Los datos globales también crean acoplamiento cuando se trata de separar el código.

Se ha hablado mucho de los beneficios de la reutilización del código. Según nuestra experiencia, esa reutilización probablemente no debería ser una preocupación primordial a la hora de crear código, pero el pensamiento de hacer que el código sea reutilizable debería ser parte de su rutina de creación de código. Cuando hace código reutilizable, le da interfaces limpias, desacoplándolo del resto de código. Esto le permite extraer un método o módulo sin arrastrar con él todo lo demás. Si su código utiliza datos globales, separarlo del resto se vuelve difícil.

Verá este problema cuando esté escribiendo pruebas unitarias para código que utiliza datos globales. Acabará escribiendo un montón de configuración para crear un entorno global solo para permitir que la prueba se ejecute.

Truco 47. Evite los datos globales.

Los datos globales incluyen *singletons*

En la sección anterior, hemos tenido cuidado de hablar de datos globales y no de variables globales. Esto se debe a que, a menudo, la gente nos dice: "¡Mirad! Ni una variable global. Lo he envuelto todo como datos de instancia en un objeto *singleton* o un módulo global".

Pruebe de nuevo, amigo. Si todo lo que tiene es un *singleton* con un puñado de variables de instancia, entonces siguen siendo solo datos globales. Simplemente tienen un nombre más largo.

Así que la gente coge este *singleton* y oculta todos los datos detrás de métodos. En vez de crear el código `Config.log_level`, ahora dicen `Config.log_level()` o `Config.getLogLevel()`. Esto es mejor, porque significa que los datos globales tienen cierta inteligencia detrás. Si decide cambiar la representación de los niveles de registro, puede mantener la compatibilidad mediante el mapeo entre lo nuevo y lo viejo en la API Config. Pero sigue teniendo solo un conjunto de datos de configuración.

Los datos globales incluyen recursos externos

Cualquier recurso externo mutable son datos globales. Si su aplicación utiliza una base de datos, un almacén de datos, un sistema de archivos, una API de servicio, etc., corre el riesgo de caer en la trampa de la globalización. De nuevo, la solución es asegurarse de que siempre envuelve estos recursos tras código que usted controla.

Truco 48. Si es lo bastante importante para ser global, envuélvalo en una API.

La herencia añade acoplamiento

El mal uso de la creación de subclases, donde una clase hereda el estado y el comportamiento de otra clase, es tan importante que hablaremos de ello en su propia sección, el tema 31, “Impuesto sobre la herencia”.

De nuevo, todo tiene que ver con el cambio

El código acoplado es difícil de cambiar: las alteraciones en una parte tienen efectos secundarios en otros lugares del código y, a menudo, en lugares difíciles de encontrar que solo salen a la luz un mes más tarde en producción.

Mantenga su código tímido: si hace que se ocupe solo de cosas de las que sepa de manera directa, mantendrá las aplicaciones desacopladas y hará que estén más abiertas al cambio.

Las secciones relacionadas incluyen

- Tema 8, “La esencia del buen diseño”.
- Tema 9, “DRY: los males de la duplicación”.
- Tema 10, “Ortogonalidad”.
- Tema 11, “Reversibilidad”.
- Tema 29, “Malabares con el mundo real”.
- Tema 30, “Transformar la programación”.
- Tema 31, “Impuesto sobre la herencia”.

- Tema 32, “Configuración”.
- Tema 33, “Romper el acoplamiento temporal”.
- Tema 34, “Estado compartido es estado incorrecto”.
- Tema 35, “Actores y procesos”.
- Tema 36, “Pizarras”.
- Hablamos de *Tell, Don’t Ask* en nuestro artículo para *Software Construction* de 2003, *The Art of Enbugging*.[2]

29 Malabares con el mundo real

Las cosas no ocurren; se hace que ocurran.

—John F. Kennedy.

Hace mucho tiempo, cuando nosotros dos todavía teníamos un atractivo juvenil, los ordenadores no eran especialmente flexibles. Por lo general, organizábamos la manera de interactuar con ellos en función de sus limitaciones.

Hoy en día, esperamos más: los ordenadores tienen que integrarse en nuestro mundo y no al revés. Y nuestro mundo es desordenado: ocurren cosas todo el tiempo, se mueve algo de un sitio a otro, cambiamos de idea... Y, de algún modo, las aplicaciones que escribimos tienen que averiguar qué hacer. Esta sección habla de la escritura de estas aplicaciones responsivas. Vamos a empezar por el concepto de “evento”.

Eventos

Un evento representa la disponibilidad de información. Podría venir del mundo exterior: un usuario que hace clic en un botón o una actualización de la cotización de las acciones. Podría ser interno: el resultado de un cálculo está listo, finaliza una búsqueda. Puede ser incluso algo tan trivial como obtener el siguiente elemento de una lista. Sea cual sea la fuente, si escribimos aplicaciones que respondan a eventos y ajustamos lo que hacen en función de esos eventos, estas aplicaciones funcionarán mejor en el mundo real. A sus usuarios les parecerán más interactivas y las propias aplicaciones harán un uso mejor de los recursos.

Pero ¿cómo podemos escribir estos tipos de aplicaciones? Sin algún tipo de estrategia, enseguida acabaremos confusos y nuestras aplicaciones serán un lío de código con acoplamiento fuerte.

Vamos a ver cuatro estrategias que ayudan.

1. Máquinas de estados finitos.
2. El patrón Observer.
3. *Publish/Subscribe*.
4. Programación reactiva y *streams*.

Máquinas de estados finitos

Dave escribe código utilizando una máquina de estados finitos (FSM, *Finite State Machine*) casi todas las semanas. Con bastante frecuencia, la implementación FSM será solo un par de líneas de código, pero esas pocas líneas ayudan a desenmarañar gran parte de desorden potencial.

Usar una FSM es fácil a nivel trivial y, aun así, muchos desarrolladores huyen de ellas. Parece existir la creencia de que son difíciles, de que solo se aplican si se trabaja con hardware o de que hace falta utilizar alguna biblioteca difícil de entender. Nada de eso es cierto.

La anatomía de una FSM pragmática

Una máquina de estados es básicamente una especificación de cómo manejar eventos. Consiste en un conjunto de estados, uno de los cuales es el estado actual. Para cada estado, hacemos una lista de los eventos que son relevantes para él. Para cada uno de esos eventos, definimos el nuevo estado actual del sistema.

Por ejemplo, podríamos estar recibiendo mensajes multiparte de un *websocket*. El primer mensaje es una cabecera. Esto va seguido de cualquier número de mensajes de datos, seguido de un mensaje de cola. Esto podría representarse (véase la figura 5.3) como una FSM.

Comenzamos en el “Estado inicial”. Si recibimos un mensaje de cabecera, pasamos al estado “Leyendo mensaje”. Si recibimos algo más mientras estamos en el estado inicial (la línea marcada con un asterisco), pasamos al estado “Error” y hemos acabado.

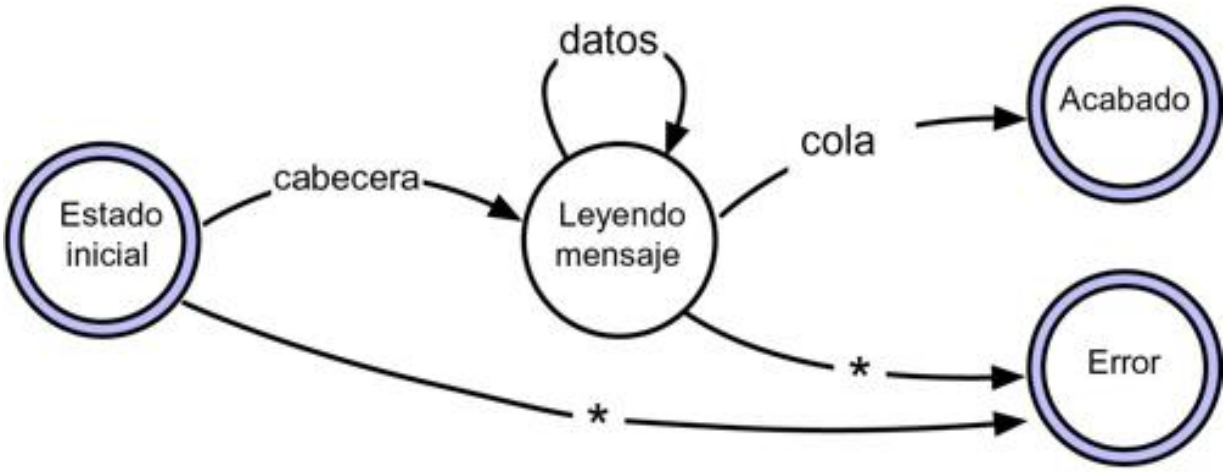


Figura 5.3.

Mientras estamos en el estado “Leyendo mensaje”, podemos aceptar mensajes de datos, en cuyo caso continuamos leyendo en el mismo estado, o podemos aceptar un mensaje de cola, que nos lleva al estado “Acabado”. Cualquier otra cosa causa una transición al estado de error.

Lo bueno de las FSM es que podemos expresarlas puramente como datos. Veamos una tabla que representa el analizador sintáctico del mensaje (figura 5.4):

Estado	Eventos			
	Cabecera	Datos	Cola	Otro
Inicial	Leyendo	Error	Error	Error
Leyendo	Error	Leyendo	Acabado	Error

Figura 5.4.

Las filas de la tabla representan los estados. Para averiguar qué hacer cuando se produce un evento, busque la fila del estado actual y la columna que representa el evento; los contenidos de esa celda son el nuevo estado.

El código que maneja esto es igual de simple:

event/simple_fsm.rb

```
Linea 1 TRANSITIONS = {
initial: {header: :reading},
```

```
reading: {data: :reading, trailer: :done},
}
5

state = :initial

while state != :done && state != :error
msg = get_next_message()
state  =  TRANSITIONS[state][msg.msg_type]  ||  :error  #  <label
id="simple-fsm-impl"/>
end
```

El código que implementa las transiciones entre estados está en la línea 10. Indexa la tabla de transiciones usando el estado actual y, después, indexa las transiciones para ese estado usando el tipo de mensaje. Si no hay ningún estado nuevo que coincida, establece el estado como `:error`.

Añadir acciones

Una FSM pura, como la que acabamos de ver, es un analizador sintáctico del flujo de eventos. Su única salida es el estado final. Podemos reforzarlo añadiendo acciones que se activen en transiciones determinadas.

Por ejemplo, podríamos necesitar extraer todas las cadenas en un archivo fuente. Una cadena es texto entre comillas, pero una barra invertida en una cadena escapa el siguiente carácter, así que `"Ignore \"quotes\""` es una sola cadena. Veamos una FSM que hace esto (figura 5.5):

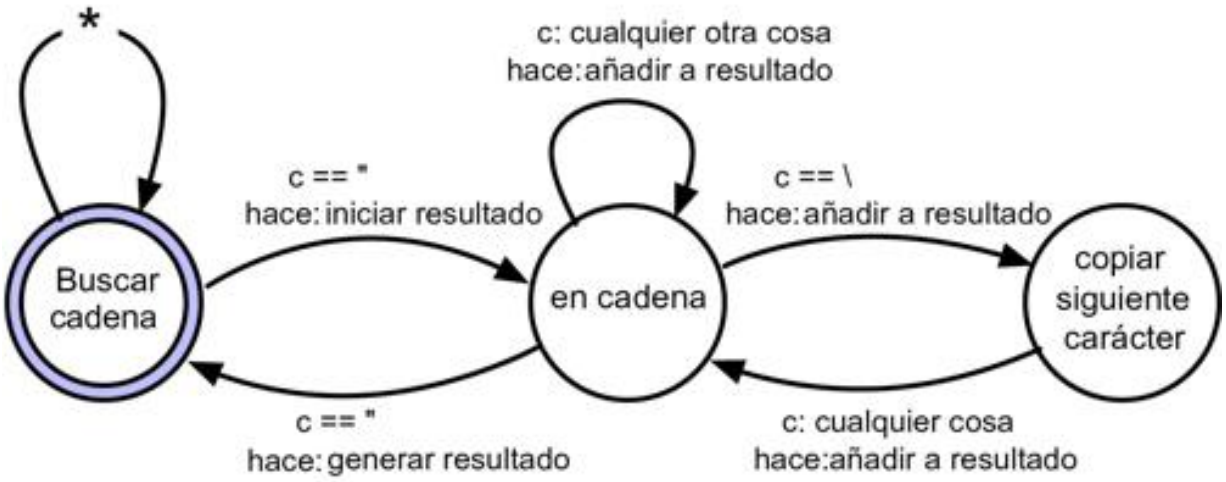


Figura 5.5.

Esta vez, cada transición tiene dos etiquetas. La superior es el evento que la desencadena y la inferior es la acción que hay que realizar a medida que nos movemos entre estados.

Vamos a expresar esto en una tabla, como hemos hecho antes. Sin embargo, en este caso cada entrada de la tabla es una lista de dos elementos que contiene el siguiente estado y el nombre de una acción:

event/strings_fsm.rb

```
TRANSITIONS = {

# actual estado nuevo acción que realizar
#————————————————————————————-

look_for_string: {
‘”’ => [ :in_string, :start_new_string ],
:default => [ :look_for_string, :ignore ],
},

in_string: {
‘”’ => [ :look_for_string, :finish_current_string ],
‘\’ => [ :copy_next_char, :add_current_to_string ],
:default => [ :in_string, :add_current_to_string ],
},

copy_next_char: {
:default => [ :in_string, :add_current_to_string ],
},
}
```

También hemos añadido la capacidad para especificar una transición predeterminada si el evento no se corresponde con ninguna de las demás transiciones de este estado. Ahora vamos a fijarnos en el código:

event/strings_fsm.rb

```
state = :look_for_string
result = []

while ch = STDIN.getc
  state, action = TRANSITIONS[state][ch] || TRANSITIONS[state]
[:default]
```

```
    case action
    when :ignore
    when :start_new_string
      result = []
    when :add_current_to_string
      result << ch
    when :finish_current_string
      puts result.join
    end
end
```

Este ejemplo es similar al anterior en el sentido de que pasamos en bucle por los eventos (los caracteres en la entrada), activando transiciones, pero hace más que el código anterior. El resultado de cada transición es tanto un estado nuevo como el nombre de una acción. Utilizamos el nombre de la acción para seleccionar el código que hay que ejecutar antes de volver a pasar por el bucle.

Este código es muy simple, pero hace el trabajo. Hay muchas otras variantes: la tabla de transiciones podría utilizar funciones anónimas o punteros de función para las acciones, podríamos envolver el código que implementa la máquina de estados en una clase separada, con su propio estado, y así sucesivamente.

No hay nada que diga que tenemos que procesar todas las transiciones de estado al mismo tiempo. Si vamos recorriendo los pasos para registrar a un usuario en nuestra aplicación, es probable que haya varias transiciones a medida que introduzcan sus datos, validen su correo electrónico, acepten los 107 avisos legales diferentes que deben ofrecer las aplicaciones ahora, etc. Mantener el estado en almacenamiento externo y utilizarlo para dirigir una máquina de estados es una manera estupenda de manejar este tipo de requisitos de flujo de trabajo.

Las máquinas de estados son un comienzo

Los desarrolladores no utilizan mucho las máquinas de estados y nos gustaría animarle a buscar oportunidades para aplicarlas, pero no resuelven todos los problemas asociados con eventos. Así pues, vamos a pasar a otras maneras de abordar los problemas de los malabares con eventos.

El patrón Observer

En el patrón Observer tenemos una fuente de eventos, denominada “observable” y una lista de clientes, los “observadores”, que están interesados en esos eventos.

Un observador registra su interés en el observable, por lo general pasando una referencia a una función a la que se debe llamar. De manera subsiguiente, cuando se produce el evento, el observable recorre su lista de observadores y llama a la función que le pasó cada uno. El evento se da como parámetro a esa llamada.

Aquí tenemos un ejemplo sencillo en Ruby. El módulo `Terminator` se utiliza para terminar la aplicación. Antes de hacerlo, sin embargo, notifica a todos sus observadores que la aplicación va a salir.[3] Podrían utilizar esta notificación para organizar recursos temporales, confirmar datos, etc.

event/observer.rb

```
module Terminator
  CALLBACKS = []

  def self.register(callback)
    CALLBACKS << callback
  end

  def self.exit(exit_status)
    CALLBACKS.each { |callback| callback.(exit_status) }
    exit!(exit_status)
  end
end

Terminator.register(-> (status) { puts “callback 1 sees #{status}” })
Terminator.register(-> (status) { puts “callback 2 sees #{status}” })

Terminator.exit(99)
$ ruby event/observer.rb
callback 1 sees 99
callback 2 sees 99
```

No hay mucho código implicado en la creación de un observable: se pasa una referencia a una función a una lista y, después, se llama a estas

funciones cuando se produce el evento. Este es un buen ejemplo de cuándo no utilizar una biblioteca.

El patrón observador/observable lleva décadas utilizándose, y nos ha funcionado bien. Predomina sobre todo en los sistemas de interfaz de usuario, donde las retrollamadas se utilizan para informar a la aplicación de que se ha producido alguna interacción.

Pero el patrón Observer tiene un problema: como cada uno de los observadores tiene que registrarse en el observable, se introduce acoplamiento. Además, puesto que en la implementación típica las retrollamadas las maneja el observable insertándolas, de forma sincrónica, puede introducir cuellos de botella en el rendimiento.

Esto se soluciona mediante la siguiente estrategia, *Publish/Subscribe*.

Publish/Subscribe

Publish/Subscribe (pubsub) generaliza el patrón Observer, resolviendo al mismo tiempo los problemas de acoplamiento y de rendimiento.

En el modelo pubsub, tenemos editores (*publishers*) y suscriptores (*subscribers*). Estos se conectan a través de canales. Los canales se implementan en un cuerpo de código aparte: a veces una biblioteca, a veces un proceso y a veces una infraestructura distribuida. Todos estos detalles de implementación se ocultan al código.

Cada canal tiene un nombre. Los suscriptores registran el interés en uno o más de esos canales con nombre y los editores escriben eventos en ellos. A diferencia del patrón Observer, la comunicación entre el editor y el suscriptor se maneja fuera del código y es potencialmente asíncrona.

Aunque podría implementar un sistema pubsub muy básico usted mismo, es probable que no le interese. La mayoría de los proveedores de servicios en la nube tienen ofertas de pubsub, lo que le permite conectar aplicaciones por todo el mundo. Cualquier lenguaje popular tendrá al menos una biblioteca pubsub.

Pubsub es una buena tecnología para desacoplar el manejo de eventos asíncronos. Permite que se añada y se sustituya código, potencialmente mientras la aplicación está ejecutándose, sin alterar el código existente. La desventaja es que puede ser difícil ver lo que está pasando en un sistema

que utiliza pubsub de forma considerable: no puede mirar un editor y ver de inmediato qué suscriptores están involucrados en un mensaje en particular.

En comparación con el patrón Observer, pubsub es un gran ejemplo de cómo se reduce el acoplamiento mediante la abstracción a través de una interfaz compartida (el canal). Sin embargo, sigue siendo en esencia solo un sistema para pasar mensajes. Crear sistemas que respondan a las combinaciones de eventos necesitará algo más que eso, así que vamos a ver maneras en las que podemos añadir una dimensión de tiempo al procesamiento de eventos.

Programación reactiva, *streams* y eventos

Si alguna vez ha utilizado una hoja de cálculo, entonces estará familiarizado con la programación reactiva. Si una celda contiene una fórmula que hace referencia a una segunda celda, actualizar esa segunda celda también hace que la primera se actualice. Los valores reaccionan cuando los valores que utilizan cambian.

Hay muchos *frameworks* que pueden ayudar con este tipo de reactividad a nivel de datos: en el reino de los navegadores, React y Vue.js son los favoritos en este momento (pero, como estamos hablando de JavaScript, esta información estará desfasada antes de que este libro se imprima siquiera).

Está claro que los eventos también pueden utilizarse para desencadenar reacciones en el código, pero eso no quiere decir que sea fácil conectarlos. Aquí es donde entran en juego los *streams*.

Los *streams* nos permiten tratar los eventos como si fuesen una colección de datos. Es como si tuviésemos una lista de eventos, que se hiciese más larga con la llegada de nuevos eventos. La belleza de esto es que podemos tratar los *streams* como cualquier otra colección: podemos manipular, combinar, filtrar y hacer todas esas cosas que sabemos hacer con datos. Incluso podemos combinar *streams* de eventos y colecciones corrientes. Y los *streams* pueden ser asíncronos, lo que significa que el código tiene la oportunidad de responder a los eventos en cuanto llegan.

El punto de referencia *de facto* actual para el manejo de eventos reactivo se define en el sitio http://reactivex.io, que define un conjunto de principios

independientes del lenguaje y documenta algunas implementaciones comunes. Aquí vamos a usar la biblioteca RxJs para JavaScript.

Nuestro primer ejemplo toma dos *streams* y los une: el resultado es un nuevo *stream* donde cada elemento contiene un ítem del primer *stream* de entrada y un ítem del otro. En este caso, el primer *stream* es solo una lista de cinco nombres de animales. El segundo *stream* es más interesante: es un temporizador de intervalos que genera un evento cada 500 ms. Como los *streams* están unidos, un resultado solo se genera cuando los datos están disponibles en ambos y, por tanto, nuestro *stream* resultante solo emite un valor cada medio segundo:

event/rx0/index.js

```
import * as Observable from 'rxjs'
import { logValues } from "../rxcommon/logger.js"

let animals = Observable.of("ant", "bee", "cat", "dog", "elk")
let ticker = Observable.interval(500)

let combined = Observable.zip(animals, ticker)
combined.subscribe(next => logValues(JSON.stringify(next)))
```

Este código utiliza una simple función de registro[4] que añade elementos a una lista en la ventana del navegador. Cada elemento lleva un sello de tiempo con el tiempo en milisegundos desde que el programa ha empezado a ejecutarse. Esto es lo que muestra para nuestro código (figura 5.6):

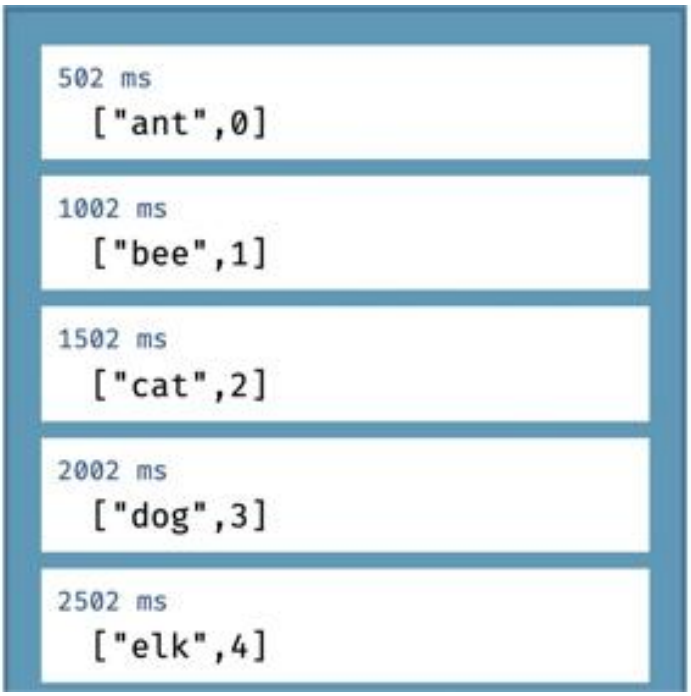


Figura 5.6.

Fíjese en los sellos de tiempo: recibimos un evento desde el *stream* cada 500 ms. Cada evento contiene un número de serie (creado por el observable `interval`) y el nombre del siguiente animal de la lista. Al verlo en directo en un navegador, las líneas de registro aparecen cada medio segundo.

Los *streams* de eventos suelen poblarse a medida que se producen los eventos, lo cual implica que los observables que los pueblan pueden ejecutarse en paralelo. Aquí hay un ejemplo que extrae información sobre los usuarios de un sitio remoto. Para esto, vamos a utilizar https://reqres.in, un sitio público que proporciona una interfaz REST abierta. Como parte de su API, podemos tomar datos de un usuario (falso) particular realizando una solicitud GET a `users/"id"`. Nuestro código extrae los usuarios con los ID 3, 2 y 1:

event/rx1/index.js

```
import * as Observable from 'rxjs'
import { mergeMap } from 'rxjs/operators'
import { ajax } from 'rxjs/ajax'
import { logValues } from "../rxcommon/logger.js"
```

```
let users = Observable.of(3, 2, 1)

let result = users.pipe(
mergeMap((user)                                                    =>
ajax.getJSON(`https://reqres.in/api/users/${user}`))
)

result.subscribe(
resp => logValues(JSON.stringify(resp.data)),
err => console.error(JSON.stringify(err))
)
```

Los detalles internos del código no son demasiado importantes. Lo interesante es el resultado, que se muestra en la siguiente imagen (figura 5.7):

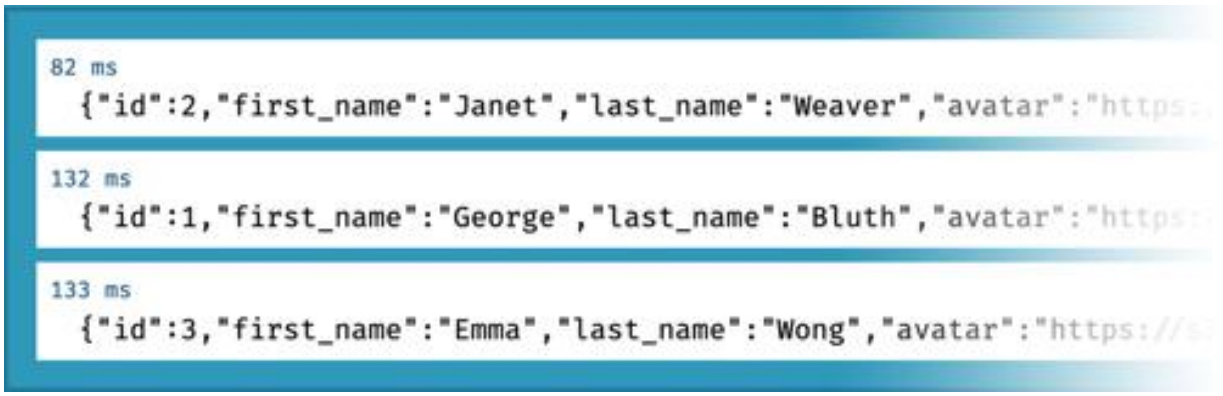


Figura 5.7.

Fíjese en los sellos de tiempo: las tres solicitudes, o los tres *streams* separados, se procesan en paralelo. El primero en volver, para id 2, ha tardado 82 ms, y los dos siguientes han vuelto 50 y 51 ms más tarde.

Los *streams* de eventos son colecciones asíncronas

En el ejemplo anterior, nuestra lista de ID de usuarios (en el observable `users`) era estática, pero no tiene por qué ser así. Quizá queramos recoger esta información cuando la gente inicie sesión en nuestro sitio. Lo único que tenemos que hacer es crear un observable evento que contenga sus ID de usuarios cuando se cree su sesión y utilizar ese observable en vez del estático. Entonces, estaríamos extrayendo detalles sobre los usuarios a

medida que recibiésemos esos ID y, presumiblemente, almacenándolos en otra parte.

Se trata de una abstracción muy potente: ya no necesitamos pensar en el tiempo como en algo que tenemos que gestionar. Los *streams* de eventos unifican el procesamiento síncrono y asíncrono detrás de una API común conveniente.

Los eventos son ubicuos

Los eventos están por todas partes. Algunos son obvios: clic en un botón, un temporizador que expira. Otros son menos evidentes: alguien que inicia sesión, una línea en un archivo que se corresponde con un patrón. Pero, sea cual sea su origen, el código que se crea en torno a eventos es más responsivo y está mejor desacoplado que su equivalente lineal.

Las secciones relacionadas incluyen

- Tema 28, “Desacoplamiento”.
- Tema 36, “Pizarras”.

Ejercicios

Ejercicio 19

En la sección sobre las FSM mencionábamos que se podía mover la implementación de la máquina de estados genérica dentro de su propia clase. Es probable que esa clase se inicializase al pasar una tabla de transiciones y un estado inicial. Pruebe a implementar el extractor de cadenas de esa manera.

Ejercicio 20

¿Cuál de estas tecnologías (quizá en combinación) sería una buena opción para las siguientes situaciones?:

- Si recibe tres eventos de caída de interfaz de red en cinco minutos, notifíqueselo al personal de operaciones.

- Si es después del atardecer, y se detecta movimiento en la parte inferior de las escaleras, seguido de movimiento detectado en la parte superior de las escaleras, active las luces del piso de arriba.
- Quiere notificar a varios sistemas de creación de informes que se ha completado un pedido.
- Para determinar si un cliente cumple los requisitos para optar a un préstamo para un coche, la aplicación necesita enviar solicitudes a tres servicios *backend* y esperar las respuestas.

30 Transformar la programación

Si no puede describir lo que está haciendo como un proceso, no sabe lo que está haciendo.

—W. Edwards Deming, (atribuido).

Todos los programas transforman datos, convirtiendo una entrada en una salida. Y, aun así, cuando pensamos en el diseño, rara vez pensamos en crear transformaciones. En vez de eso, nos preocupamos por clases y módulos, estructuras de datos y algoritmos, lenguajes y *frameworks*.

Creemos que poner el foco en el código de esta manera a menudo pasa por alto lo importante: necesitamos volver a pensar en los programas como en algo que transforma entradas en salidas. Cuando lo hacemos, muchos de los detalles que antes nos preocupaban se evaporan sin más. La estructura se vuelve más clara, el manejo de errores es más coherente y se reduce mucho el acoplamiento.

Para empezar nuestra investigación, vamos a coger la máquina del tiempo para viajar a los setenta y pedir a un programador de Unix que nos escriba un programa que haga una lista de los cinco archivos más largos de un árbol de directorios, donde “más largo” significa “que tiene el mayor número de líneas”.

Podrá esperar que el programador vaya a un editor y empiece a teclear en C, pero no lo haría, porque está pensando en esto en relación a lo que tenemos (un árbol de directorios) y lo que queremos (una lista de archivos). Iría a un terminal y escribiría algo como:

```
$ find . -type f | xargs wc -l | sort -n | tail -5
```

Se trata de una serie de transformaciones:

- `find . -type f`
  Escribe una lista de todos los archivos (`-type f`) en o debajo del directorio actual (`.`) a una salida estándar.
- `xargs wc -l`
  Lee líneas de la entrada estándar y hace que todas ellas se pasen como argumentos al comando `wc -l`. El programa `wc` con la opción `-l` cuenta el número de líneas en cada uno de sus argumentos y escribe cada resultado como "contar nombre de archivo" para la salida estándar.
- `sort -n`
  Ordena la entrada estándar asumiendo que cada línea empieza con un número (`-n`), escribiendo el resultado para la salida estándar.
- `tail -5`
  Lee la entrada estándar y escribe solo las cinco últimas líneas para la salida estándar.

Si ejecutamos esto en el directorio de nuestro libro, obtenemos

```
470 ./test_to_build.pml
487 ./dbc.pml
719 ./domain_languages.pml
727 ./dry.pml
9561 total
```

Esa última línea es el número total de líneas de todos los archivos (no solo de los que se muestran), porque eso es lo que hace `wc`. Podemos quitarla solicitando una línea más de `tail` y, después, ignorando esa última línea:

```
$ find . -type f | xargs wc -l | sort -n | tail -6 | head -5
   470 ./debug.pml
   470 ./test_to_build.pml
   487 ./dbc.pml
   719 ./domain_languages.pml
   727 ./dry.pml
```

Vamos a ver esto en lo que respecta a los datos que fluyen entre los pasos individuales. Nuestro requisito original, “el top 5 de los archivos en relación al número de líneas”, se convierte en una serie de transformaciones (como se muestra en la figura 5.8).

nombre del directorio
▶ lista de archivos
▶ lista con números de líneas
▶ lista ordenada
▶ cinco más altos + total
▶ cinco más altos

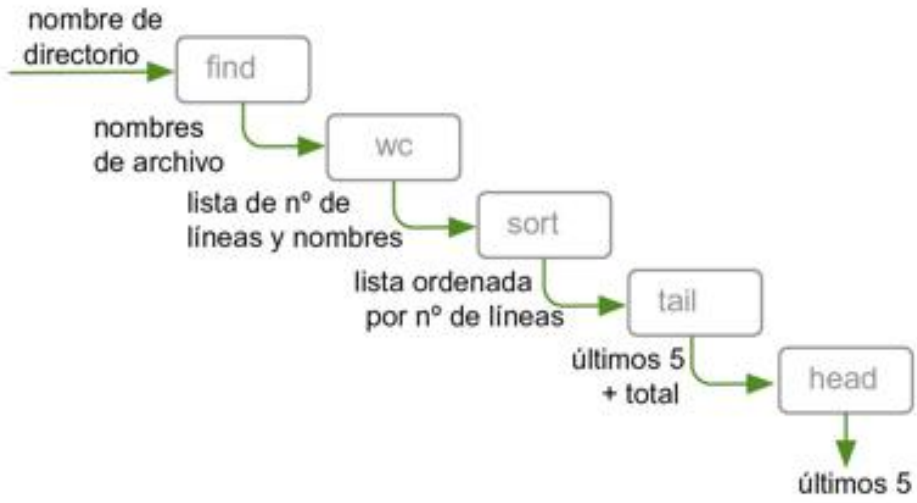


Figura 5.8. La *pipeline* `find` como una serie de transformaciones.

Es casi como una cadena de montaje industrial: se introducen datos brutos por un extremo y el producto acabado (información) sale por el otro.

Y nos gusta pensar en todo el código de esa forma.

Truco 49. La programación trata sobre el código, pero los programas tratan sobre los datos.

Encontrar transformaciones

A veces, la manera más sencilla de encontrar las transformaciones es empezar por los requisitos y determinar sus entradas y salidas. Ahora hemos definido la función que representa el programa general. Después, podemos encontrar pasos que nos lleven desde la entrada a la salida. Este es un enfoque de arriba abajo.

Por ejemplo, queremos crear un sitio web para gente que juega a juegos de palabras que encuentre todas las palabras que pueden formarse con un conjunto de letras. Aquí la entrada es un conjunto de letras y la salida es una lista de palabras de tres letras, de cuatro letras, etc.:

“lvyin” se transforma en ▶ 3 => ivy, lin, nil, yin
4 => inly, liny, viny
5 => vinyl

(Sí, todas son palabras, al menos según el diccionario en inglés de macOS).

El truco detrás de la aplicación general es simple: tenemos un diccionario que agrupa palabras por una firma, elegida de manera que todas las palabras que tengan las mismas letras tendrán la misma firma. La función de firma más simple es solo la lista ordenada de letras de la palabra. Entonces, podemos buscar una cadena de entrada mediante la generación de una firma para ella y, después, viendo qué palabras (si las hay) en el diccionario tienen esa misma firma. Así pues, el buscador de anagramas se descompone en cuatro transformaciones separadas:

Paso	Transformación	Datos de muestra
Paso 0:	Entrada inicial	"ylvin"
Paso 1:	Todas las combinaciones de tres o más letras	vin, viy, vil, vny, vnl, vyl, iny, inl, iyl, nyl, viny, vinl, viyl, vnyl, inyl, vinyl
Paso 2:	Firmas de combinaciones	inv, ivy, ilv, nvy, lnv, lvy, iny, iln, ily, lny, invy, ilnv, ilvy, lnvy, ilny, ilnvy
Paso 3:	Lista de todas las palabras del diccionario que coinciden con cualquiera de las firmas	ivy, yin, nil, lin, viny, liny, inly, vinyl
Paso 4:	Palabras agrupadas por longitud	3 => ivy, lin, nil, yin 4 => inly, liny, viny 5 => vinyl

Transformaciones hasta abajo

Vamos a empezar por fijarnos en el paso 1, que toma una palabra y crea una lista de todas las combinaciones de tres o más letras. Este paso en sí puede expresarse como una lista de transformaciones:

Paso	Transformación	Datos de muestra
Paso 1.0:	Entrada inicial	"vinyl"
Paso 1.1:	Convertir en caracteres	v, i, n, y, l
Paso 1.2:	Obtener todos los subconjuntos	[], [v], [i], ... [v,i], [v,n], [v,y], ... [v,i,n], [v,i,y], ... [v,n,y,l], [i,n,y,l], [v,i,n,y,l]
Paso 1.3:	Solo aquellos de más de tres caracteres	[v,i,n], [v,i,y], ... [i,n,y,l], [v,i,n,y,l]
Paso 1.4:	Volver a convertir en cadenas	[vin,viy, ... inyl,vinyl]

Ahora hemos llegado al punto en el que podemos implementar con facilidad cada transformación en código (en este caso, usando Elixir):

function-pipelines/anagrams/lib/anagrams.ex

```
defp all_subsets_longer_than_three_characters(word) do
  word
  |> String.codepoints()
  |> Comb.subsets()
  |> Stream.filter(fn subset -> length(subset) >= 3 end)
  |> Stream.map(&List.to_string(&1))
end
```

¿Qué pasa con el operador |>?

Elixir, junto con muchos otros lenguajes funcionales, tiene un operador *pipeline*, llamado a veces *forward pipe* o solo *pipe*.[5] Lo que hace es tomar el valor a su izquierda e insertarlo como el primer parámetro de la función a su derecha, así que:

```
“vinyl” |> String.codepoints |> Comb.subsets()
```

es lo mismo que escribir:

```
Comb.subsets(String.codepoints(“vinyl”))
```

(Otros lenguajes podrían introducir este valor usado por la *pipe* como el último valor de la función siguiente; depende mucho del estilo de las bibliotecas integradas).

Podría pensar que esto no es más que azúcar sintáctico, pero, de una manera muy real, el operador *pipeline* es una oportunidad revolucionaria para pensar de manera diferente. Utilizar una *pipeline* significa que está pensando automáticamente en lo que respecta a la transformación de datos; cada vez que ve `|>` en realidad está viendo un lugar donde los datos fluyen entre una transformación y la siguiente. Muchos lenguajes tienen algo similar: Elm, F# y Swift tienen `|>`, Clojure tiene `->` y `->>` (que funcionan de manera un poco diferente), R tiene %>%. Haskell tiene operadores *pipe* y también hace que sea fácil declarar unos nuevos. En el momento de escribir este libro, se está hablando de añadir `|>` a JavaScript. Si su lenguaje actual soporta algo similar, está de suerte. Si no, consulte el cuadro “El lenguaje X no tiene *pipelines*” que veremos enseguida. De momento, volvamos al código.

Seguir transformando...

Vamos a fijarnos ahora en el paso 2 del programa principal, donde convertimos subconjuntos en firmas. De nuevo, se trata de una transformación simple; una lista de subconjuntos se convierte en una lista de firmas:

Paso	Transformación	Datos de muestra
Paso 2.0:	entrada inicial	vin, viy, ... inyl, vinyl
Paso 2.1:	convertir en firmas	inv, ivy ... ilny, inlvy

El código de Elixir en el siguiente listado es igual de simple:

function-pipelines/anagrams/lib/anagrams.ex

```
defp as_unique_signatures(subsets) do
  subsets
```

```
  |> Stream.map(&Dictionary.signature_of/1)
end
```

Ahora transformamos esa lista de firmas: cada firma se asigna a la lista de palabras conocidas con la misma firma o `nil` si no hay tales palabras. Después, tenemos que eliminar `nils` y aplanar las listas anidadas en un único nivel:

function-pipelines/anagrams/lib/anagrams.ex

```
defp find_in_dictionary(signatures) do
  signatures
  |> Stream.map(&Dictionary.lookup_by_signature/1)
  |> Stream.reject(&is_nil/1)
  |> Stream.concat(&(&1))
end
```

El paso 4, el agrupamiento de las palabras por longitud, es otra transformación simple, que convierte nuestra lista en un mapa donde las claves son las longitudes y los valores son todas las palabras con esa longitud:

function-pipelines/anagrams/lib/anagrams.ex

```
defp group_by_length(words) do
  words
  |> Enum.sort()
  |> Enum.group_by(&String.length/1)
end
```

Juntarlo todo

Hemos escrito cada una de las transformaciones individuales. Ahora es el momento de encadenarlas todas para formar nuestra función principal:

function-pipelines/anagrams/lib/anagrams.ex

```
def anagrams_in(word) do
  word
  |> all_subsets_longer_than_three_characters()
  |> as_unique_signatures()
```

```
   |> find_in_dictionary()
   |> group_by_length()
end
```

¿Funciona? Vamos a probar:

```
iex(1)> Anagrams.anagrams_in “lyvin”
%{
   3 => [“ivy”, “lin”, “nil”, “yin”],
   4 => [“inly”, “liny”, “viny”],
   5 => [“vinyl”]
}
```

El lenguaje X no tiene *pipelines*

Las *pipelines* llevan mucho tiempo utilizándose, pero solo en lenguajes de nicho. Hace poco que se han incorporado a la corriente principal, y muchos lenguajes populares todavía no tienen soporte para el concepto.
La buena noticia es que pensar en transformaciones no requiere una sintaxis de lenguaje en particular: es más una filosofía de diseño. Aún construimos el código como transformaciones, pero lo escribimos como una serie de asignaciones:

```
const content = File.read(file_name);
const lines = find_matching_lines(content, pattern)
const result = truncate_lines(lines)
```

Es un poco más tedioso, pero hace el trabajo.

¿Por qué es esto tan genial?

Fijémonos de nuevo en el cuerpo de la función principal:

```
word
|> all_subsets_longer_than_three_characters()
|> as_unique_signatures()
|> find_in_dictionary()
|> group_by_length()
```

Es solo una cadena de transformaciones necesarias para cumplir nuestro requisito, cada una de las cuales toma la entrada de la transformación anterior y pasa la salida a la siguiente. Esto es lo más parecido a un código literario que podemos conseguir.

Pero también hay algo más profundo. Si su formación está relacionada con la programación orientada a objetos, entonces sus reflejos le exigirán que oculte los datos, encapsulándolos dentro de objetos. Después, estos objetos van de un lado a otro, cambiando el estado de los demás. Esto introduce mucho acoplamiento y es una razón de peso por la que los sistemas orientados a objetos son difíciles de cambiar.

Truco 50. No acumule estados; vaya pasándolos.

En los modelos transformacionales, le damos la vuelta a eso. En vez de estanques de datos diseminados por todo el sistema, piense en los datos como un río poderoso, algo que fluye. Los datos se convierten en un par de la funcionalidad: una *pipeline* es una secuencia de código ▶ datos ▶ código ▶ datos.... Los datos ya no están atados a un grupo concreto de funciones, como lo están en la definición de una clase, sino que son libres para representar el progreso de nuestra aplicación a medida que transforman sus entradas en sus salidas. Esto significa que podemos reducir mucho el acoplamiento: una función puede utilizarse (y reutilizarse) en cualquier parte donde sus parámetros coincidan con la salida de alguna otra función.

Sí, sigue habiendo cierto grado de acoplamiento, pero, según nuestra experiencia, es más fácil de gestionar que con el estilo orientado a objetos del mando y control. Y, si está utilizando un lenguaje con comprobación de tipos, recibirá alertas en tiempo de compilación cuando intente conectar dos cosas incompatibles.

¿Qué pasa con el manejo de errores?

Hasta ahora, nuestras transformaciones han funcionado en un mundo en el que nada va mal. Pero ¿cómo podemos utilizarlas en el mundo real? Si solo podemos crear cadenas lineales, ¿cómo podemos añadir toda esa lógica condicional que necesitamos para la comprobación de errores?

Hay muchas formas de hacerlo, pero todas dependen de una convención básica: nunca pasamos valores brutos entre transformaciones, sino que los envolvemos en una estructura (o tipo) de datos que también nos dice si el

valor contenido es válido. En Haskell, por ejemplo, este envoltorio se llama `Maybe`. En F# y Scala es `Option`.

El modo de utilizar esto es específico de cada lenguaje. No obstante, en general, hay dos maneras básicas de escribir el código: puede gestionar la comprobación de errores dentro de sus transformaciones o fuera de ellas.

Elixir, que es lo que hemos utilizado hasta ahora, no tiene este soporte integrado. Para nuestro propósito, es algo bueno, ya que así podemos mostrar una implementación desde cero. Algo similar debería funcionar en la mayoría del resto de lenguajes.

Primero, elija una representación

Necesitamos una representación para nuestro envoltorio (la estructura de datos que lleva a todas partes un valor o una indicación de error). Puede utilizar estructuras para esto, pero Elixir ya tiene una convención bastante fuerte: las funciones tienden a devolver una tupla que contiene o `{:ok, value}` o `{:error, reason}`. Por ejemplo, `File.open` devuelve, bien `:ok` y un proceso E/S, bien `:error` y un código de motivo:

```
iex(1)> File.open("/etc/passwd")
{:ok, #PID<0.109.0>}
iex(2)> File.open("/etc/wombat")
{:error, :enoent}
```

Vamos a utilizar la tupla `:ok/:error` como nuestro envoltorio cuando pasemos cosas a través de una *pipeline*.

Después, gestiónela dentro de cada transformación

Vamos a escribir una función que devuelva todas las líneas de un archivo que contengan una cadena dada, truncada a los primeros 20 caracteres. Queremos escribirla como una transformación, así que la entrada será un nombre de archivo y una cadena que coincida, y la salida será, bien una tupla `:ok` con una lista de líneas, bien una tupla `:error` con algún tipo de motivo. La función de nivel superior debería parecerse a esto:

function-pipelines/anagrams/lib/grep.ex

```
def find_all(file_name, pattern) do
  File.read(file_name)
  |> find_matching_lines(pattern)
  |> truncate_lines()
end
```

Aquí no hay una comprobación de errores explícita, pero, si cualquier paso de la *pipeline* devuelve una tupla de error, entonces la *pipeline* devolverá ese error sin ejecutar las funciones que siguen.[6] Hacemos esto utilizando la coincidencia de patrones de Elixir:

function-pipelines/anagrams/lib/grep.ex

```
defp find_matching_lines({:ok, content}, pattern) do
  content
  |> String.split(~r/\n/)
  |> Enum.filter(&String.match?(&1, pattern))
  |> ok_unless_empty()
end

defp find_matching_lines(error, _), do: error

# ————

defp truncate_lines({ :ok, lines }) do
  lines
  |> Enum.map(&String.slice(&1, 0, 20))
  |> ok()
end

defp truncate_lines(error), do: error

# ————

defp ok_unless_empty([]), do: error("nothing found")
defp ok_unless_empty(result), do: ok(result)

defp ok(result), do: { :ok, result }
defp error(reason), do: { :error, reason }
```

Eche un vistazo a la función `find_matching_lines`. Si su primer parámetro es una tupla `:ok`, utiliza el contenido de esa tupla para encontrar líneas que coincidan con el patrón. Sin embargo, si el primer parámetro no es una tupla `:ok`, la segunda versión de la función se ejecuta, lo que devuelve ese parámetro. De este modo, la función reenvía sin más un error

hacia abajo en la *pipeline*. Lo mismo se aplica a `truncate_lines`. Podemos jugar con esto en la consola:

```
iex> Grep.find_all “/etc/passwd”, ~r/www/
{:ok, [“_www:*:70:70:World W”, “_wwwproxy:*:252:252:”]}
iex> Grep.find_all “/etc/passwd”, ~r/wombat/
{:error, “nothing found”}
iex> Grep.find_all “/etc/koala”, ~r/www/
{:error, :enoent}
```

Puede ver que un error en cualquier parte de la *pipeline* se convierte de inmediato en el valor de la *pipeline*.

O gestiónela en la *pipeline*

Puede que esté mirando las funciones `find_matching_lines` y `truncate_lines` pensando que hemos movido la carga del manejo de errores a las transformaciones. Tendría razón. En un lenguaje que utiliza la coincidencia de patrones en llamadas a funciones, como Elixir, el efecto se reduce, pero sigue siendo feo.

Estaría bien que Elixir tuviese una versión del operador *pipeline* `|>` que conociese las tuplas `:ok/:error` y que cortocircuitase la ejecución cuando se produjese un error.[7] Pero el hecho es que no nos permite añadir nada similar y, en cierto modo, eso es aplicable a varios lenguajes más.

El problema al que nos enfrentamos es que cuando se produce un error no queremos ejecutar código que avance más por la *pipeline*, y que no queremos que ese código sepa lo que está pasando. Eso significa que necesitamos posponer la ejecución de las funciones de la *pipeline* hasta que sepamos que los pasos anteriores en la *pipeline* han tenido éxito. Para hacer eso, necesitaremos cambiarlas de llamadas a funciones a valores de funciones a los que se pueda llamar más adelante. Aquí hay una implementación:

function-pipelines/anagrams/lib/grep1.ex

```
defmodule Grep1 do

  def and_then({ :ok, value }, func), do: func.(value)
  def and_then(anything_else, _func), do: anything_else
```

```
  def find_all(file_name, pattern) do
File.read(file_name)
    |> and_then(&find_matching_lines(&1, pattern))
    |> and_then(&truncate_lines(&1))
  end

  defp find_matching_lines(content, pattern) do
    content
|> String.split(~r/\n/)
    |> Enum.filter(&String.match?(&1, pattern))
    |> ok_unless_empty()
  end

  defp truncate_lines(lines) do
    lines
|> Enum.map(&String.slice(&1, 0, 20))
    |> ok()
  end

  defp ok_unless_empty([]), do: error("nothing found")
  defp ok_unless_empty(result), do: ok(result)

  defp ok(result), do: { :ok, result }
  defp error(reason), do: { :error, reason }
end
```

La función `and_then` es un ejemplo de función *bind*: toma un valor envuelto en algo y, después, aplica una función a ese valor, devolviendo un nuevo valor envuelto. Usar la función `and_then` en la *pipeline* requiere algo de puntuación extra porque Elixir necesita que se le ordene que convierta llamadas de funciones en valores de funciones, pero ese esfuerzo extra se ve compensado por el hecho de que las funciones que realizan las transformaciones se simplifican: cada una toma solo un valor (y cualquier parámetro extra) y devuelve `{:ok, new_value}` o `{:error, reason}`.

Las transformaciones transforman la programación

Pensar en el código como una serie de transformaciones (anidadas) puede ser un enfoque liberador a la hora de programar. Lleva un tiempo acostumbrarse a ello, pero, una vez que desarrolle el hábito, verá que su

código se vuelve más limpio, sus funciones más cortas y sus diseños más planos. Pruébelo.

Las secciones relacionadas incluyen

- Tema 8, “La esencia del buen diseño”.
- Tema 17, “Jugar con el intérprete de comandos”.
- Tema 26, “Cómo equilibrar los recursos”.
- Tema 28, “Desacoplamiento”.
- Tema 35, “Actores y procesos”.

Ejercicio

Ejercicio 21

¿Puede expresar los siguientes requisitos como una transformación de nivel superior? Es decir, para cada uno, identifique la entrada y la salida.

1. Los gastos de envío y el impuesto sobre las ventas se añaden a un pedido.
2. Su aplicación carga información de configuración desde un archivo con nombre.
3. Alguien inicia sesión en una aplicación web.

Ejercicio 22

Ha identificado la necesidad de validar y convertir un campo de entrada de una cadena a un entero entre 18 y 150. La transformación general se describe mediante:

```
contenidos del campo como cadena
    ▶ [validar & convertir]
      ▶ {:ok, valor} | {:error, motivo}
```

Escriba las transformaciones individuales que constituyen “validar” y “convertir”.

Ejercicio 23

En el cuadro “El lenguaje X no tiene *pipelines*”, escribimos:

```
const content = File.read(file_name);
const lines = find_matching_lines(content, pattern)
const result = truncate_lines(lines)
```

Mucha gente escribe código orientado a objetos encadenando llamadas a métodos y podría verse tentada de escribir esto como algo similar a:

```
const result = content_of(file_name)
       .find_matching_lines(pattern)
       .truncate_lines()
```

¿Cuál es la diferencia entre estos dos fragmentos de código? ¿Cuál cree que preferimos nosotros?

31 Impuesto sobre la herencia

> *Querías un plátano, pero lo que conseguiste fue un gorila sujetando el plátano y la jungla completa.*
>
> *—Joe Armstrong.*

¿Programa en un lenguaje orientado a objetos? ¿Utiliza herencias?
¡Si lo hace, pare! Es probable que eso no sea lo que le interese hacer.
Veamos por qué.

Algo de historia

La herencia apareció por primera vez en Simula 67 en 1969. Era una solución elegante para el problema de poner en cola múltiples tipos de eventos en la misma lista. El enfoque de Simula era utilizar algo denominado “clases prefijo”. Podíamos escribir algo como esto:

```
link CLASS car;
   ... implementation of car

link CLASS bicycle;
   ... implementation of bicycle
```

Aquí `link` es una clase prefijo que añade la funcionalidad de las listas vinculadas. Esto nos permite añadir tanto coches (*cars*) como bicicletas (*bicycles*) a la lista de cosas que pueden estar esperando en (por ejemplo) un

semáforo. En la terminología actual, `link` sería una clase principal o padre.

El modelo mental utilizado por los programadores de Simula fue que los datos de instancia y la implementación de la clase `link` se anteponían a la implementación de las clases `car` y `bicycle`. La parte de `link` se veía casi como si fuese un contenedor que llevase de un lado a otro coches y bicicletas. Esto les daba una forma de polimorfismo: tanto coches como bicicletas implementaban la interfaz `link` porque ambos contenían el código `link`.

Después de Simula llegó Smalltalk. Alan Kay, uno de los creadores de Smalltalk, describe en una respuesta en Quora[8] en 2019 por qué Smalltalk tiene herencia:

> *Así que cuando diseñé Smalltalk-72 (y fue una broma para divertirme mientras pensaba en Smalltalk-71), pensé que sería divertido utilizar sus dinámicas parecidas a Lisp para hacer experimentos con la "programación diferencial" (es decir: varias formas de lograr "esto es como aquello, excepto").*

Esto es una creación de subclases puramente por comportamiento.

Estos dos estilos de herencia (que en realidad tenían bastante en común) se desarrollaron a lo largo de las décadas siguientes. El enfoque de Simula, que sugería que la herencia era una manera de combinar tipos, continuó en lenguajes como C++ y Java. La escuela de Smalltalk, donde la herencia era una organización dinámica de comportamientos, se veía en lenguajes como Ruby y JavaScript.

Así pues, ahora nos encontramos con una generación de usuarios del desarrollo orientado a objetos que utilizan la herencia por una de dos razones: no les gusta teclear o les gustan los tipos.

Aquellos a los que no les gusta teclear protegen sus dedos utilizando la herencia para añadir funcionalidad común desde una clase base a clases secundarias: la clase `User` y la clase `Product` son subclases de `ActiveRecord::Base`.

Aquellos a los que les gustan los tipos utilizan la herencia para expresar la relación entre clases: `Car` es-un-tipo-de `Vehicle`.

Por desgracia, ambos tipos de herencia tienen problemas.

Problema de usar la herencia al compartir código

La herencia es acoplamiento. No solo está la clase secundaria acoplada a la principal, la principal de la principal, etc., sino que el código que utiliza la secundaria también está asociado a todos sus ancestros. Veamos un ejemplo:

```
class Vehicle
     def initialize
          @speed = 0
     end
   def stop
          @speed = 0
   end
        def move_at(speed)
          @speed = speed
     end
end

class Car < Vehicle
     def info
          “I’m car driving at #{@speed}”
     end
end

# código de nivel superior
my_ride = Car.new
my_ride.move_at(30)
```

Cuando el nivel superior llama a `my_ride.move_at`, el método al que se invoca está en `Vehicle`, padre de `Car`.

Ahora el desarrollador a cargo de `Vehicle` modifica la API, así que `move_at` se convierte en `set_velocity`, y la variable de instancia `@speed` se convierte en `@velocity`.

Se espera que un cambio en la API rompa clientes de la clase `Vehicle`, pero no que lo haga el nivel superior: por lo que a él respecta, está utilizando `Car`. Lo que hace la clase `Car` en lo relativo a la implementación no es asunto del código de nivel superior, pero, aun así, se rompe.

De manera similar, el nombre de una variable de instancia es puramente un detalle de implementación interna, pero, cuando `Vehicle` cambia,

también rompe (de forma silenciosa) `Car`.

Demasiado acoplamiento.

Problemas de usar la herencia para construir tipos

Algunas personas ven la herencia como una manera de definir tipos nuevos. Su diagrama de diseño favorito muestra jerarquías de clases. Ven los problemas de la misma manera que los caballeros científicos victorianos veían la naturaleza, como algo que había que descomponer en categorías (figura 5.9).

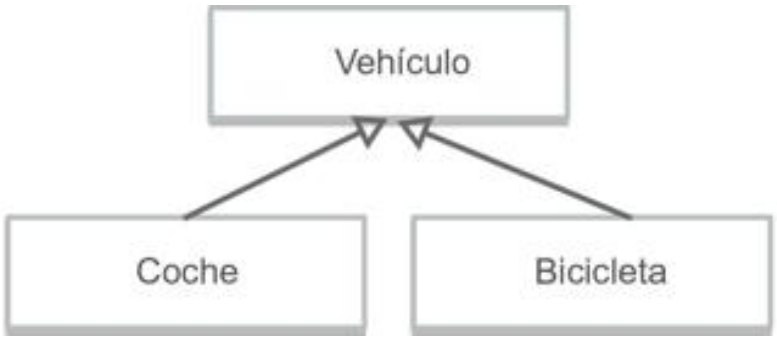


Figura 5.9.

Por desgracia, estos diagramas pronto crecen hasta convertirse en monstruosidades que cubren paredes enteras, en los que se añade capa tras capa para expresar hasta el más mínimo matiz de diferenciación entre clases. Esta complejidad añadida puede hacer que la aplicación sea más frágil, ya que los cambios pueden propagarse hacia arriba y hacia abajo por muchas capas (figura 5.10).

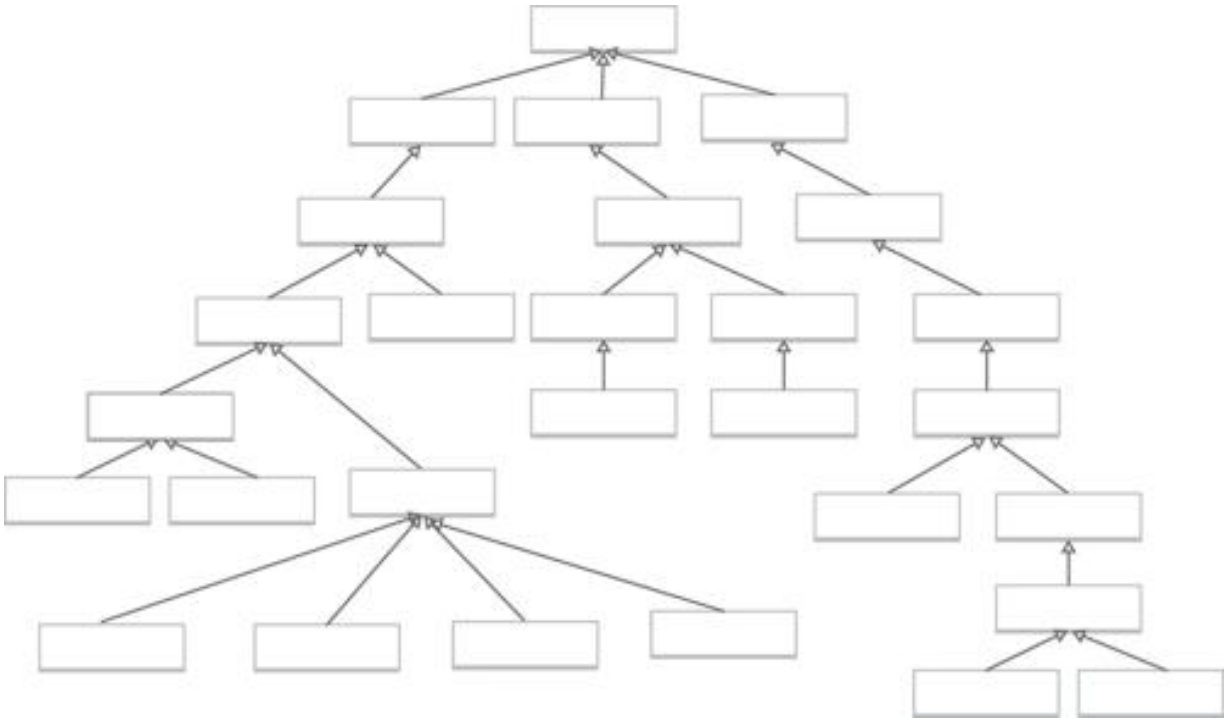


Figura 5.10.

Lo que es incluso peor, sin embargo, es el problema de la herencia múltiple. `Car` puede ser un tipo de `Vehicle`, pero también puede ser un tipo de `Asset`, `InsuredItem`, `LoanCollateral`, etc. Modelar esto de forma correcta necesitaría herencia múltiple.

C++ dio mala fama a la herencia múltiple en los noventa debido a cierta semántica de desambiguación cuestionable. Como resultado, muchos lenguajes orientados a objetos actuales no la ofrecen, así que, incluso si está satisfecho con los árboles de tipos complejos, no podrá modelar su dominio de manera precisa en cualquier caso.

Truco 51. No pague el impuesto sobre la herencia.

Las alternativas son mejores

Permítanos sugerir tres técnicas que implican que nunca más debería necesitar utilizar la herencia de nuevo:

- Interfaces y protocolos.

- Delegación.
- *Mixins* y *traits*.

Interfaces y protocolos

La mayoría de los lenguajes orientados a objetos nos permiten especificar que una clase implementa uno o más conjuntos de comportamientos. Podría decirse, por ejemplo, que una clase `Car` implementa el comportamiento `Drivable` y el comportamiento `Locatable`. La sintaxis utilizada para hacer esto varía; en Java, podría tener este aspecto:

```
public class Car implements Drivable, Locatable {
   // Código para la clase Car. Este código debe incluir
   // la funcionalidad de Drivable
   // y de Locatable
}
```

`Drivable` y `Locatable` son lo que Java llama interfaces; otros lenguajes lo llaman protocolos y algunos lo llaman *traits* (aunque no es lo que vamos a llamar *trait* más adelante).

Las interfaces se definen así:

```
public interface Drivable {
   double getSpeed();
   void stop();
}

public interface Locatable() {
   Coordinate getLocation();
   boolean locationIsValid();
}
```

Estas declaraciones no crean código; simplemente dicen que cualquier clase que implemente `Drivable` debe implementar los dos métodos `getSpeed` y `stop`, y una clase que sea `Locatable` debe implementar `getLocation` y `locationIsValid`. Esto significa que nuestra anterior definición de clase de `Car` solo será válida si incluye estos cuatro métodos.

Lo que hace que las interfaces y los protocolos sean tan potentes es que podemos utilizarlos como tipos, y cualquier clase que implemente la interfaz apropiada será compatible con ese tipo. Si tanto `Car` como `Phone` implementan `Locatable`, podríamos almacenar ambas en una lista de elementos localizables:

```
List<Locatable> items = new ArrayList<>();
items.add(new Car(...));
items.add(new Phone(...));
items.add(new Car(...));
// ...
```

Entonces, podemos procesar esa lista con la tranquilidad de saber que todos los elementos tienen `getLocation` y `locationIsValid`:

```
void printLocation(Locatable item) {
   if (item.locationIsValid() {
      print(item.getLocation().asString());
}

// ...

items.forEach(printLocation);
```

Truco 52. Decántese por las interfaces para expresar polimorfismo.

Las interfaces y los protocolos nos proporcionan polimorfismo sin herencia.

Delegación

La herencia anima a los desarrolladores a crear clases cuyos objetos tengan una gran cantidad de métodos. Si una clase principal tiene 20 métodos y la subclase solo quiere hacer uso de dos de ellos, sus objetos aún tendrán los otros 18 por ahí y se les podrá llamar. La clase ha perdido el control de su interfaz. Se trata de un problema común; muchos *frameworks* de interfaz de usuario y de persistencia insisten en que los componentes de la aplicación conviertan en subclase alguna clase base proporcionada:

```
class Account < PersistenceBaseClass
```

```
end
```

Ahora, la clase `Account` lleva de un lado a otro toda la API de la clase de persistencia. En vez de eso, imagine una alternativa que utilice la delegación, como en el siguiente ejemplo:

```
class Account
  def initialize(. . .)
    @repo = Persister.for(self)
  end

  def save
    @repo.save()
  end
end
```

Ahora no exponemos nada de la API del *framework* a los clientes de nuestra clase `Account`: ahora ese desacoplamiento se ha roto. Pero hay más. Ahora que ya no estamos limitados por la API del *framework* que estamos utilizando, somos libres de crear la API que necesitamos. Sí, podríamos hacer eso antes, pero siempre correríamos el riesgo de que la interfaz que escribiésemos pudiese ignorarse y en su lugar se utilizase la API de persistencia. Ahora lo controlamos todo.

Truco 53. Delegue en los servicios: “tiene un” triunfa sobre “es un”.

De hecho, podemos llevar esto un paso más allá. ¿Por qué `Account` tiene que saber cómo persistir? ¿No es su trabajo conocer y hacer cumplir las reglas de negocio de la cuenta?

```
class Account
  # nada salvo cosas de la cuenta
end

class AccountRecord
  # envuelve una cuenta con la capacidad
  # de ser obtenida y almacenada
end
```

Ahora estamos desacoplados de verdad, pero ha tenido un precio. Estamos teniendo que escribir más código y, por lo general, parte de él sería

repetitivo: es probable que todas nuestras clases de registro necesiten un método `find` por ejemplo.

Por suerte, eso es lo que hacen por nosotros los *mixins* y los *traits*.

Mixins, *traits*, categorías, extensiones de protocolo...

Como industria, nos encanta poner nombre a las cosas. Con bastante frecuencia, ponemos muchos nombres a la misma cosa. ¿Cuántos más mejor?

A eso nos enfrentamos cuando nos fijamos en los *mixins*. La idea básica es simple: queremos ser capaces de extender clases y objetos con nueva funcionalidad sin usar la herencia. Así pues, creamos un conjunto de estas funciones, ponemos un nombre a ese conjunto y después, de algún modo, extendemos una clase u objeto con ellas. En ese punto, hemos creado una nueva clase u objeto nuevo que combina las capacidades de la original y todos sus *mixins*. En la mayoría de los casos, podremos llevar a cabo esta extensión incluso si no tenemos acceso al código fuente de la clase que estamos extendiendo.

Ahora la implementación y el nombre de esta característica varían entre lenguajes. Aquí tendemos a llamarlos *mixins*, pero en realidad queremos que piense en esto como una característica que no es específica de un solo lenguaje. Lo importante es la capacidad que tienen todas estas implementaciones: fusionar la funcionalidad entre cosas existentes y cosas nuevas.

Como ejemplo, vamos a volver a nuestro ejemplo de `AccountRecord`. Tal y como lo hemos dejado, `AccountRecord` necesitaba conocer ambas cuentas y nuestro *framework* de persistencia. También necesitaba delegar todos los métodos en la capa de persistencia que quería exponer al mundo exterior.

Los *mixins* nos dan una alternativa. Primero, podríamos escribir un *mixin* que implemente (por ejemplo) dos de tres de los métodos estándar del buscador. Después, podríamos añadirlos a `AccountRecord` como un *mixin*. Y, a medida que escribimos nuevas clases para cosas persistentes, podemos añadirles también un *mixin*:

```
mixin CommonFinders {
```

```
  def find(id) { ... }
  def findAll() { ... }
end

class AccountRecord extends BasicRecord with CommonFinders
class OrderRecord extends BasicRecord with CommonFinders
```

Podemos llevar esto mucho más lejos. Por ejemplo, todos sabemos que nuestros objetos de negocio necesitan código de validación para evitar que se infiltren datos malos en nuestros cálculos. Pero ¿a qué nos referimos exactamente con “validación”?

Si tomamos una cuenta, por ejemplo, es probable que haya muchas capas de validación diferentes que podrían aplicarse:

- Validar que una contraseña con *hash* coincide con la introducida por el usuario.
- Validar datos de formulario introducidos por el usuario cuando se creó una cuenta.
- Validar datos de formulario introducidos por un administrador al actualizar la información del usuario.
- Validar datos añadidos a la cuenta por otros componentes del sistema.
- Validar la consistencia de los datos antes de su persistencia.

Un enfoque común (y creemos que no es el ideal) es agrupar todas las validaciones en una sola clase (el objeto de negocio/objeto de persistencia) y, después, añadir banderas para controlar qué se activa en qué circunstancias.

Creemos que una manera mejor es utilizar *mixins* para crear clases especializadas para situaciones apropiadas:

```
class AccountForCustomer extends Account
  with AccountValidations,AccountCustomerValidations

class AccountForAdmin extends Account
  with AccountValidations,AccountAdminValidations
```

Aquí, ambas clases derivadas incluyen validaciones comunes a todos los objetos de la cuenta. La variante del cliente también incluye validaciones apropiadas para las API de cara al cliente, mientras que la variante del

administrador contiene las validaciones del administrador (seguramente menos restrictivas). Ahora, al pasar instancias de `AccountForCustomer` o `AccountForAdmin` de un lado a otro, nuestro código garantiza de manera automática que se aplica la validación correcta.

Truco 54. Utilice *mixins* para compartir funcionalidad.

La herencia rara vez es la respuesta

Hemos echado un vistazo rápido a tres alternativas a la herencia de clases tradicional:

- Interfaces y protocolos.
- Delegación
- *Mixins* y *traits*

Cada uno de estos métodos puede convenirle más en diferentes circunstancias, dependiendo de si su objetivo es compartir información de tipos, añadir funcionalidad o compartir métodos. Como ocurre con todo en la programación, intente utilizar la técnica que mejor exprese su intención. E intente no llevarse toda la jungla al viaje.

Las secciones relacionadas incluyen

- Tema 8, “La esencia del buen diseño”.
- Tema 10, “Ortogonalidad”.
- Tema 28, “Desacoplamiento”.

Retos

- La próxima vez que se vea creando subclases, tómese un minuto para examinar las opciones. ¿Puede conseguir lo que quiere con interfaces, delegación o *mixins*? ¿Puede reducir el acoplamiento al hacerlo?

32 Configuración

Deja que todas tus cosas tengan su sitio; deja que todos tus asuntos tengan su momento.

—Benjamin Franklin, Trece virtudes, autobiografía.

Cuando el código depende de valores que pueden cambiar cuando la aplicación pase a estar disponible para su uso, mantenga esos valores externos a la aplicación. Cuando la aplicación vaya a ejecutarse en entornos diferentes, y de forma potencial para clientes diferentes, mantenga los valores específicos del entorno y del cliente fuera de la aplicación. De este modo, estará parametrizando su aplicación; el código se adapta a los lugares en los que se ejecuta.

Truco 55. Parametrice su aplicación utilizando configuración externa.

Las cosas comunes que es probable que quiera poner en datos de configuración incluyen:

- Credenciales para servicios externos (base de datos, API de terceros, etc.).
- Destinos y niveles de registro.
- Puerto, dirección IP, máquina y nombres de clústeres que utiliza la aplicación.
- Parámetros de validación específicos del entorno.
- Parámetros establecidos de manera externa, como tasas impositivas.
- Detalles de formato específicos del sitio.
- Claves de licencia.

Básicamente, busque cualquier cosa que sepa que tendrá que cambiar que se exprese fuera del cuerpo principal del código y póngalo en un *bucket* de configuración.

Configuración estática

Muchos *frameworks*, y bastantes aplicaciones personalizadas, mantienen la configuración en archivos planos o tablas de base de datos. Si la

información está en archivos planos, la tendencia es utilizar algún formato de texto simple listo para usar. En la actualidad, YAML y JSON son opciones populares para esto. A veces, aplicaciones escritas en lenguajes de *scripts* utilizan archivos de código fuente para fines especiales, dedicados a contener solo la configuración. Si la información está estructurada y es probable que el cliente la cambie (las tasas impositivas de las ventas, por ejemplo), podría ser mejor almacenarla en una tabla de base de datos. Y, por supuesto, puede usar ambas cosas, dividiendo la información de la configuración en función del uso.

Sea cual sea la forma que use, la configuración se lee en la aplicación como una estructura de datos, por lo general cuando se inicia la aplicación. Normalmente, esta estructura de datos se hace global, con el razonamiento de que eso hace que sea más fácil para cualquier parte del código obtener los valores que alberga.

Preferimos que no haga eso. En su lugar, envuelva la información de la configuración en una API (fina). Esto desacopla su código de los detalles de la representación de configuración.

Configuración como servicio

Aunque la configuración estática es habitual, en la actualidad nosotros preferimos un enfoque diferente. Todavía queremos que los datos de configuración se mantengan externos a la aplicación, pero, en vez de en un archivo plano o una base de datos, nos gustaría verlos almacenados tras una API de servicio. Esto tiene varios beneficios:

- Múltiples aplicaciones pueden compartir información de configuración, con autenticación y control de acceso para limitar lo que puede ver cada una.
- Los cambios en la configuración pueden realizarse a nivel global.
- Los datos de configuración pueden mantenerse a través de una interfaz de usuario especializada.
- Los datos de configuración se vuelven dinámicos.

Este último punto, el de que la configuración debería ser dinámica, es crucial a medida que avanzamos hacia aplicaciones de alta disponibilidad.

La idea de que deberíamos detener y reiniciar una aplicación para cambiar un solo parámetro no está en sintonía con la realidad moderna. Al usar un servicio de configuración, los componentes de la aplicación podrían registrarse para recibir notificaciones de actualizaciones en parámetros que utilizan, y el servicio podría enviarles mensajes que contuviesen los valores nuevos si se cambiasen y cuando se cambiasen.

Tomen la forma que tomen, los datos de configuración dirigen el comportamiento en tiempo de ejecución de una aplicación. Cuando los valores de configuración cambian, no es necesario reconstruir el código.

No escriba código dodo

Sin configuración externa, el código no es tan adaptable ni flexible como debería. ¿Es eso malo? Bueno, en el mundo real, las especies que no se adaptan mueren.

El dodo no se adaptó a la presencia de los humanos y su ganado en la isla Mauricio y se extinguió enseguida.[9] Se trata de la primera extinción documentada de una especie por la mano del hombre. No deje que su proyecto (o su carrera) acabe como el dodo.

Figura 5.11.

Las secciones relacionadas incluyen

- Tema 9, “DRY: los males de la duplicación”.
- Tema 14, “Lenguajes de dominio”.
- Tema 16, “El poder del texto simple”.
- Tema 28, “Desacoplamiento”.

No se exceda

En la primera edición de este libro, sugeríamos utilizar configuración en lugar de código de una manera similar, pero parece que deberíamos haber sido un poco más específicos en nuestras explicaciones. Cualquier consejo puede llevarse al extremo o usarse de forma inadecuada, así que veamos algunas advertencias:
No se exceda. Uno de nuestros primeros clientes decidió que todos y cada uno de los campos de su aplicación debería ser configurable. Como resultado, se necesitaban semanas para hacer incluso los cambios más pequeños, porque había que implementar tanto el campo como el código de administrador para guardarlo y editarlo. Tenían unas 40.000 variables de configuración y una pesadilla de código entre manos.
No pase decisiones a configuración por pereza. Si hay un debate genuino acerca de si una característica debería funcionar de esta manera o de aquella, o si debería ser elección de los usuarios, pruébela de una manera y reciba *feedback* acerca de si la decisión ha sido buena.

---

[1] Bueno, en realidad no es una ley. Es más bien "la idea genial de Demeter".

[2] https://media.pragprog.com/articles/jan_03_enbug.pdf.

[3] Sí, sabemos que Ruby ya tiene esta capacidad con su función `at_exit`.

[4] https://media.pragprog.com/titles/tpp20/code/event/rxcommon/logger.js.

[5] Parece que el primer uso de los caracteres `|>` como *pipe* data de 1994, en una discusión sobre el lenguaje Isobelle/ML, archivada en https://blogs.msdn.microsoft.com/dsyme/2011/05/17/archeological-semiotics-the-birth-of-the-pipeline-symbol-1994/.

[6] Aquí nos hemos tomado cierta libertad. Técnicamente, sí ejecutamos las siguientes funciones. Simplemente no ejecutamos el código dentro de ellas.

[7] De hecho, podría añadir un operador así a Elixir utilizando su servicio de macros; un ejemplo de esto es la biblioteca Monad en hexadecimal. Podría utilizar la construcción `with` de Elixir, pero entonces pierde gran parte del sentido de escribir transformaciones que consigue con las *pipelines*.

[8] https://www.quora.com/What-does-Alan-Kay-think-about-inheritance-in-object- oriented-programming.

[9] No ayudó que los colonos golpeasen a estos tranquilos (es decir: estúpidos) pájaros por diversión.

6
Concurrencia

Para que estemos todos de acuerdo, vamos a empezar por algunas definiciones:

- La concurrencia es cuando en la ejecución dos o más porciones de código actúan como si se ejecutasen al mismo tiempo. El paralelismo es cuando se ejecutan al mismo tiempo.
- Para tener concurrencia, necesitamos ejecutar código en un entorno que pueda cambiar la ejecución entre diferentes partes del código cuando está ejecutándose. A menudo, esto se implementa mediante el uso de fibras, hilos y procesos.
- Para tener paralelismo, necesitamos hardware que pueda hacer dos cosas a la vez. Podrían ser múltiples núcleos en una CPU, múltiples CPU en un ordenador o múltiples ordenadores conectados entre sí.

Todo es concurrente

Es imposible escribir código en un sistema de tamaño decente que no tenga aspectos concurrentes. Puede que sean explícitos o que estén enterrados dentro de una biblioteca. La concurrencia es un requisito si quiere que su aplicación sea capaz de lidiar con el mundo real, donde las cosas son asíncronas: los usuarios están interactuando, están extrayéndose datos, se está llamando a servicios externos, todo al mismo tiempo. Si fuerza a este proceso a que se produzca en serie, donde primero pasa una cosa, luego otra, y así sucesivamente, el sistema dará la sensación de ir muy lento y es probable que no saque el máximo partido a la potencia del hardware en el que se ejecuta.

En este capítulo, vamos a ver la concurrencia y el paralelismo.

A menudo, los desarrolladores hablan de acoplamiento entre porciones de código. Se refieren a las dependencias y a cómo esas dependencias hacen que las cosas sean difíciles de cambiar. Pero hay otra forma de acoplamiento. El acoplamiento temporal se produce cuando el código impone una secuencia en las cosas que no es necesaria para resolver el problema que está tratándose. ¿Depende de que el “tic” vaya antes del “toc”? No si quiere mantener la flexibilidad. ¿Accede su código a múltiples servicios *back-end* de forma secuencial, uno tras otro? No si quiere mantener a sus clientes. En “Romper el acoplamiento temporal”, veremos maneras de identificar este tipo de acoplamiento temporal.

¿Por qué es tan difícil escribir código concurrente y paralelo? Una razón es que aprendimos a programar utilizando sistemas secuenciales y nuestros lenguajes tienen características que son relativamente seguras cuando se usan de manera secuencial, pero se convierten en un lastre una vez que pueden ocurrir dos cosas al mismo tiempo. Uno de los mayores culpables de esto es el estado compartido. Eso no significa solo variables globales: cada vez que dos o más porciones de código albergan referencias a la misma porción de datos mutables, tenemos estado compartido. Y “Estado compartido es estado incorrecto”. La sección describe varias soluciones alternativas para eso, pero al final todas son propensas a errores.

Si eso le entristece, ¡no desespere! Hay formas mejores de construir aplicaciones concurrentes. Una de ellas es utilizar el modelo actor, donde procesos independientes, que no comparten datos, se comunican a través de canales utilizando una semántica simple y definida. Hablaremos sobre la teoría y la práctica de este enfoque en “Actores y procesos”.

Por último, echaremos un vistazo a las “Pizarras”. Se trata de sistemas que actúan como una combinación de un almacén de objetos y un intermediario *publish/subscribe* inteligente. En su forma original, nunca llegaron a despegar del todo, pero hoy en día cada vez se ven más implementaciones de capas de middleware con semántica de estilo pizarra. Si se usan correctamente, estos tipos de sistemas ofrecen una cantidad importante de desacoplamiento.

El código concurrente y paralelo solía ser algo exótico. Ahora es necesario.

33 Romper el acoplamiento temporal

Puede que esté preguntándose: “¿De qué va el desacoplamiento temporal?”. Va sobre el tiempo. A menudo, el tiempo es un aspecto ignorado de las arquitecturas de software. El único tiempo que nos preocupa es el del calendario, el tiempo que tenemos hasta el envío, pero aquí no estamos hablando de eso, sino del papel del tiempo como elemento de diseño del propio software. Hay dos aspectos del tiempo que son importantes para nosotros: la concurrencia (cosas que ocurren al mismo tiempo) y el orden (las posiciones relativas de las cosas en el tiempo).

Por lo general, no abordamos la programación con ninguno de estos aspectos en mente. Cuando la gente se sienta por primera vez a diseñar una arquitectura o escribir un programa, las cosas tienden a ser lineales. Esa es la manera en la que piensa la mayoría de la gente; se hace esto y, después, siempre se hace aquello. Pero pensar de este modo conduce al acoplamiento temporal: un acoplamiento en el tiempo. Siempre debe llamarse al método `A` antes que al método `B`; solo puede ejecutarse un informe cada vez; hay que esperar a que se recargue la pantalla antes de que se reciba el clic del botón. Tic debe venir antes que toc.

Este enfoque no es muy flexible, y tampoco es muy realista.

Necesitamos permitir la concurrencia y pensar en el desacoplamiento cada vez u ordenar dependencias. Al hacerlo, podemos ganar flexibilidad y reducir cualquier dependencia basada en el tiempo en muchas áreas del desarrollo: análisis del flujo de trabajo, arquitectura, diseño y despliegue. El resultado serán sistemas sobre los que es más fácil razonar, con potencial para responder con más rapidez y fiabilidad.

Buscar la concurrencia

En muchos proyectos, necesitamos modelar y analizar los flujos de trabajo de las aplicaciones como parte del diseño. Nos gustaría averiguar lo que puede ocurrir al mismo tiempo y lo que debe pasar siguiendo un orden estricto. Una manera de hacerlo es capturar el flujo de trabajo utilizando una notación como el diagrama de actividad.[1]

Truco 56. Analice el flujo de trabajo para mejorar la concurrencia.

Un diagrama de actividad consiste en un conjunto de acciones dibujadas como cajas redondeadas. La flecha que sale de una acción lleva, bien a otra acción (que puede comenzar una vez que se completa la primera acción), bien a una línea gruesa denominada barra de sincronización. Una vez que se completan todas las acciones que llevan a la barra de sincronización, podemos continuar a lo largo de cualquier flecha que salga de la barra. Una acción que no tenga ninguna flecha que conduzca a ella puede comenzarse en cualquier momento.

Podemos utilizar diagramas de actividad para maximizar el paralelismo identificando actividades que podrían realizarse en paralelo, pero no lo hacen.

Por ejemplo, podríamos estar escribiendo el software para una máquina robótica que prepare piña colada. Se nos dice que los pasos son:

1. Abrir licuadora
2. Abrir preparado para piña colada
3. Echar preparado en licuadora
4. Medir 1/2 taza de ron blanco
5. Verter ron
6. Añadir 2 tazas de hielo
7. Cerrar licuadora
8. Licuar durante 1 minuto
9. Abrir licuadora
10. Sacar vasos
12. Servir
11. Sacar sombrillas rosas

Sin embargo, un camarero perdería su trabajo si siguiese estos pasos, uno a uno, en orden. Incluso aunque estas acciones se describen en serie, muchas de ellas podrían realizarse en paralelo. Usaremos el siguiente diagrama de actividad (figura 6.1) para capturar y razonar acerca de la concurrencia potencial. Puede resultar muy revelador ver dónde existen en realidad las dependencias. En este ejemplo, las tareas del nivel superior (1, 2, 4, 10 y 11) pueden ocurrir de forma concurrente, al principio. Las tareas

3, 5 y 6 pueden ocurrir en paralelo más tarde. Si está en un concurso de preparación de piñas coladas, estas optimizaciones marcarán la diferencia.

Oportunidades para la concurrencia

Los diagramas de actividad muestran las áreas potenciales de concurrencia, pero no dicen nada acerca de si vale la pena explotar estas áreas. Por ejemplo, en el caso de la piña colada, un camarero necesitaría cinco manos para poder ejecutar todas las tareas iniciales potenciales a la vez.

Y aquí es donde entra la parte del diseño. Cuando echamos un vistazo a las actividades, nos damos cuenta de que la número 8, licuar, llevará un minuto. Durante ese tiempo, nuestro camarero puede sacar los vasos y las sombrillitas (actividades 10 y 11) y, probablemente, tener tiempo para servir a otro cliente.

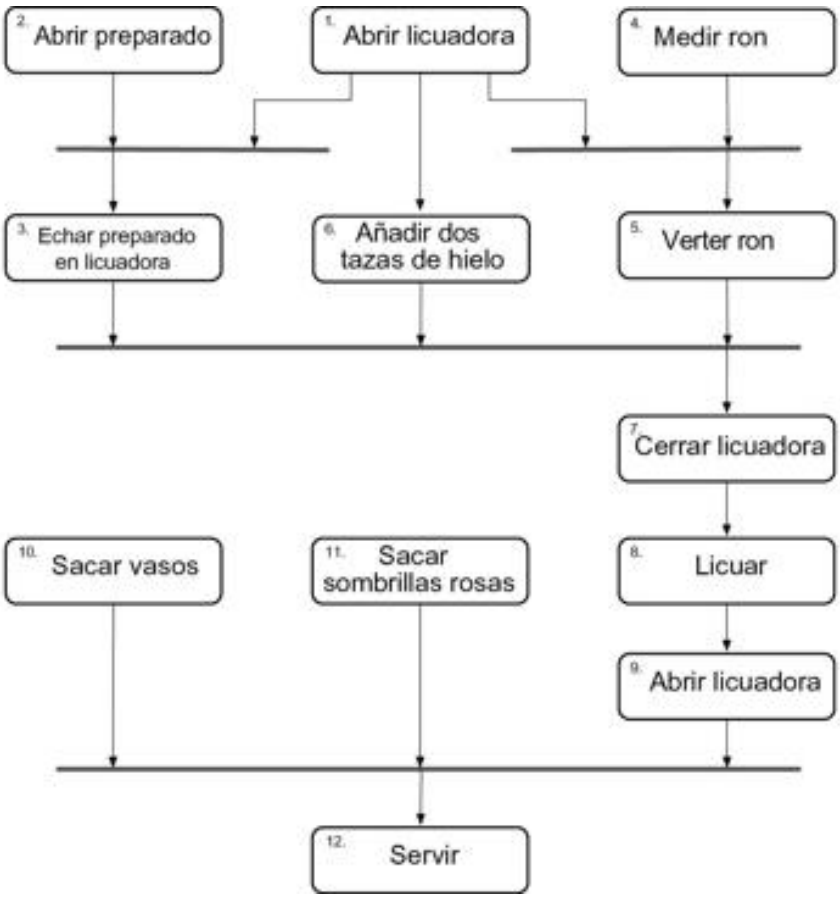


Figura 6.1.

Aplicar formato con mayor rapidez

Este libro está escrito en texto simple. Para crear la versión para su impresión, para un libro electrónico o para lo que sea, ese texto se introduce a través de una *pipeline* de procesadores. Algunos buscan construcciones específicas (citas bibliográficas, entradas para el índice alfabético, marcado especial para los trucos, etc.). Otros procesadores operan sobre el documento en conjunto. Muchos de los procesadores en la *pipeline* tienen que acceder a información externa (lectura de archivos, escritura de archivos, canalización a través de programas externos). Todo este trabajo a velocidad relativamente lenta nos da la oportunidad de explotar la concurrencia: de hecho, cada paso en la *pipeline* se ejecuta de forma concurrente, leyendo del paso anterior y escribiendo en el siguiente.

Además, algunas partes del procesador requieren un uso relativamente intensivo del procesador. Una de ellas es la conversión de las fórmulas matemáticas. Por distintas razones históricas, cada ecuación puede tardar hasta 500 ms en convertirse. Para acelerar las cosas, sacamos provecho del paralelismo. Como cada fórmula es independiente de las demás, convertimos cada una en su propio proceso paralelo y llevamos los resultados de vuelta al libro a medida que están disponibles.

Como resultado, el libro se construye mucho mucho más rápido en máquinas con múltiples núcleos.

(Y sí, lo cierto es que descubrimos varios errores de concurrencia en nuestra *pipeline* por el camino...).

Y eso es lo que estamos buscando cuando diseñamos para tener concurrencia. Esperamos encontrar actividades que lleven tiempo, pero no tiempo en el código. Consultar una base de datos, acceder a un servicio externo, esperar la entrada del usuario: por lo general, todas estas cosas paralizarían nuestro sistema hasta que se completasen. Y todas ellas son oportunidades para hacer algo más productivo que el equivalente en CPU a esperar de brazos cruzados.

Oportunidades para el paralelismo

Recuerde la distinción: la concurrencia es un mecanismo de software y el paralelismo es una cuestión de hardware. Si tenemos múltiples procesadores, ya sea locales o remotos, si podemos dividir el trabajo entre ellos, podemos reducir el tiempo total que tardan las cosas.

Las cosas ideales para dividir de este modo son las partes del trabajo que son relativamente independientes, donde cada componente puede proceder sin tener que esperar a que los demás hagan algo. Un patrón común es tomar una parte grande del trabajo, dividirla en porciones independientes, procesar cada una en paralelo y, después, combinar los resultados.

Un ejemplo interesante de esta práctica es la manera en que funciona el compilador para el lenguaje Elixir. Cuando empieza, divide el proyecto que está construyendo en módulos y compila cada uno en paralelo. A veces un módulo depende de otro, en cuyo caso su compilación se detiene hasta que los resultados de la construcción del otro módulo estén disponibles. Cuando el módulo del nivel superior se completa, eso significa que todas las dependencias se han compilado. El resultado es una compilación rápida que saca provecho de todos los núcleos disponibles.

Identificar oportunidades es la parte fácil

Volvamos a sus aplicaciones. Hemos identificado lugares en los que se beneficiarían de la concurrencia y el paralelismo. Ahora viene la parte complicada: cómo implementarlo de manera segura. Ese es el tema del resto del capítulo.

Las secciones relacionadas incluyen

- Tema 10, “Ortogonalidad”.
- Tema 26, “Cómo equilibrar los recursos”.
- Tema 28, “Desacoplamiento”.
- Tema 36, “Pizarras”.

Retos

- ¿Cuántas tareas realiza en paralelo mientras se prepara para ir al trabajo por la mañana? ¿Podría expresar esto en un diagrama de actividad UML? ¿Podría encontrar alguna manera de prepararse más deprisa aumentando la concurrencia?

34 Estado compartido es estado incorrecto

Está en su restaurante favorito. Ha terminado los platos principales y pregunta al camarero si queda algo de tarta de manzana. Él se da la vuelta, ve un trozo en el expositor y dice que sí. Usted lo pide y suspira de felicidad.

Mientras tanto, en el otro lado del restaurante, otro cliente hace la misma pregunta a su camarera, ella se da la vuelta, confirma que hay un trozo y el cliente lo pide.

Uno de los clientes va a llevarse una decepción.

Cambie el expositor por una cuenta bancaria conjunta y a los camareros por dispositivos de punto de venta. Usted y su pareja deciden comprar un teléfono nuevo al mismo tiempo, pero en la cuenta solo hay dinero suficiente para uno. Alguien (el banco, la tienda, usted) va a estar muy descontento.

Truco 57. Un estado compartido es un estado incorrecto.

El problema es el estado compartido. Cada camarero del restaurante ha mirado al expositor sin tener en cuenta al otro. Cada dispositivo de punto de venta ha mirado la cuenta del banco sin tener en cuenta al otro.

Actualizaciones no atómicas

Vamos a mirar nuestro ejemplo del restaurante como si fuese código (véase la figura 6.2).

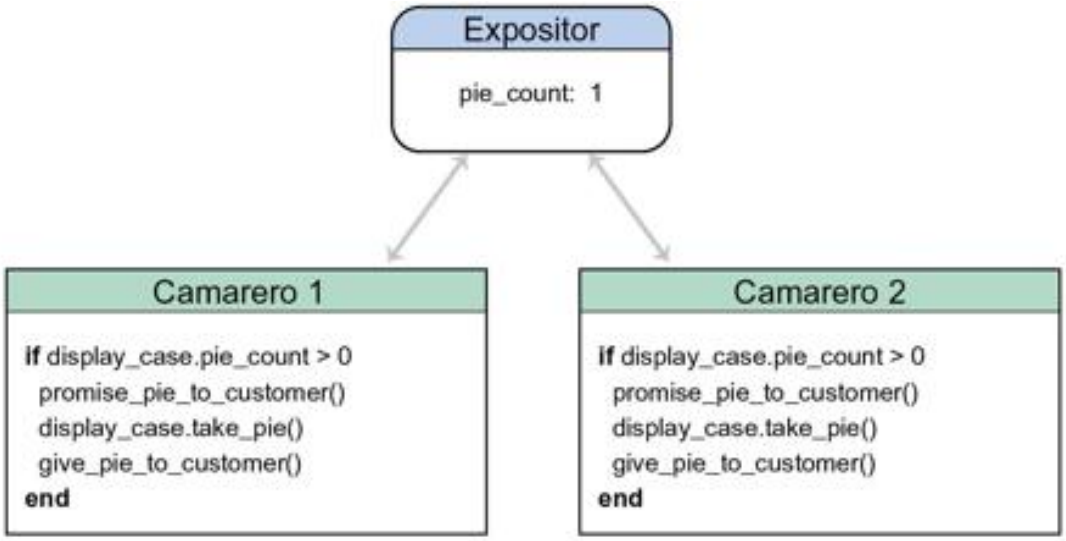


Figura 6.2.

Los dos camareros operan de manera concurrente (y, en la vida real, en paralelo). Vamos a fijarnos en su código:

```
if display_case.pie_count > 0
   promise_pie_to_customer()
   display_case.take_pie()
   give_pie_to_customer()
end
```

El camarero 1 obtiene el total de tarta (`pie_count`) actual, y descubre que es 1. Promete la tarta al cliente (`promise_pie_to_customer`). Pero, en ese punto, se ejecuta la camarera 2. También ve que el total de tarta es uno y hace la misma promesa a su cliente. Entonces, uno de los dos toma el último trozo de tarta (`take_pie`) y el otro camarero introduce algún tipo de error de estado (que probablemente implica mucha humillación).

Aquí, el problema no es que dos procesos puedan escribir en la misma memoria. El problema es que ninguno de los procesos puede garantizar que su vista de esa memoria es consistente. En efecto, cuando un camarero ejecuta `display_case.pie_count()`, copia el valor del expositor en su propia memoria. Si el valor en el expositor cambia, su memoria (que está utilizando para tomar decisiones) se queda desactualizada.

Todo esto se debe a que obtener y después actualizar el total de tarta no es una operación atómica: el valor subyacente puede cambiar en el medio.

Así pues, ¿cómo podemos hacerla atómica?

Semáforos y otras formas de exclusión mutua

Un semáforo es simplemente una cosa que solo una persona puede poseer cada vez. Puede crear un semáforo y, después, utilizarlo para controlar el acceso a algún otro recurso. En nuestro ejemplo, podríamos crear un semáforo para controlar el acceso al expositor de la tarta y adoptar la convención de que cualquiera que quiera actualizar el contenido de ese expositor solo puede hacerlo si sostiene ese semáforo. Supongamos que el restaurante decide arreglar el problema de la tarta con un semáforo físico. Colocan un duende de plástico en el expositor de la tarta. Antes de que cualquier camarero pueda vender la tarta, debe tener el duende en la mano. Una vez que se ha completado su pedido (lo que significa servir la tarta en la mesa), pueden devolver el duende a su sitio como guardián del tesoro de las tartas, listo para interceder en el siguiente pedido.

Vamos a ver esto en código. En el estilo clásico, la operación para coger el semáforo se llamaba P y la operación para soltarlo se llamaba V.[2] Hoy en día utilizamos términos como bloquear/desbloquear, reclamar/liberar, etc.

```
case_semaphore.lock()

if display_case.pie_count > 0
   promise_pie_to_customer()
   display_case.take_pie()
   give_pie_to_customer()
end

case_semaphore.unlock()
```

Este código asume que ya se ha creado y almacenado un semáforo en la variable `case_semaphore`. Vamos a asumir que ambos camareros ejecutan el código al mismo tiempo. Ambos intentan bloquear el semáforo, pero solo uno lo consigue. El que obtiene el semáforo continúa ejecutándose con normalidad. El que no lo consigue se queda suspendido hasta que el semáforo vuelve a estar disponible (el camarero espera...). Cuando el primer camarero completa el pedido, desbloquea el semáforo y el segundo camarero continúa ejecutándose. Ahora ve que no hay tarta en el expositor y se disculpa con el cliente.

Hay algunos problemas con este enfoque. Es probable que el más significativo sea que solo funciona porque todos los que acceden al

expositor de la tarta están de acuerdo en la convención de usar el semáforo. Si a alguien se le olvida (es decir, si algún desarrollador escribe código que no sigue esa convención), volvemos al caos.

Haga que el recurso sea transaccional

El diseño actual es pobre porque delega la responsabilidad de proteger el acceso al expositor de la tarta en las personas que lo utilizan. Vamos a cambiarlo para centralizar ese control. Para ello, tenemos que cambiar la API de manera que los camareros comprueben la cantidad y también cojan un trozo de tarta en una sola llamada:

```
slice = display_case.get_pie_if_available()
if slice
   give_pie_to_customer()
end
```

Para hacer que esto funcione, necesitamos escribir un método que se ejecute como parte del propio expositor:

```
def get_pie_if_available() ####
   if @slices.size > 0 #
      update_sales_data(:pie) #
      return @slices.shift #
   else # ¡código incorrecto!
      false #
   end #
end ####
```

Este código ilustra una confusión habitual. Hemos movido el acceso al recurso a un lugar central, pero todavía se puede llamar a nuestro método desde múltiples hilos concurrentes, así que necesitamos protegerlo con un semáforo:

```
def get_pie_if_available()
   @case_semaphore.lock()

   if @slices.size > 0
      update_sales_data(:pie)
      return @slices.shift
   else
      false
   end
```

```
  @case_semaphore.unlock()
end
```

Incluso este código podría no ser correcto. Si `update_sales_data` lanza una excepción, el semáforo nunca se desbloqueará, y todos los accesos futuros al expositor se quedarán esperando de manera indefinida. Necesitamos gestionar esto:

```
def get_pie_if_available()
  @case_semaphore.lock()

  try {
    if @slices.size > 0
      update_sales_data(:pie)
      return @slices.shift
      else
        false
    end
  }
  ensure {
    @case_semaphore.unlock()
  }
end
```

Puesto que este es un error muy común, muchos lenguajes ofrecen bibliotecas que gestionan esto por nosotros:

```
def get_pie_if_available()
@case_semaphore.protect() {
if @slices.size > 0
update_sales_data(:pie)
return @slices.shift
else
false
end
}
end
```

Transacciones de recursos múltiples

Nuestro restaurante acaba de instalar un congelador para helado. Si un cliente pide tarta con una bola de helado, el camarero tendrá que comprobar que hay tanto tarta como helado disponibles.

Podríamos cambiar el código del camarero por algo como:

```
slice = display_case.get_pie_if_available()
scoop = freezer.get_ice_cream_if_available()

if slice && scoop
   give_order_to_customer()
end
```

Sin embargo, esto no funcionará. ¿Qué pasa si reclamamos un trozo de tarta, pero cuando intentamos conseguir la bola de helado descubrimos que no hay? Ahora nos encontramos con que tenemos un trozo de tarta con el que no podemos hacer nada (porque nuestro cliente debe tener helado). Y el hecho de que tengamos la tarta significa que no está en el expositor, así que no está disponible para cualquier otro cliente que (siendo un purista) no la quiera con helado.

Podríamos arreglar esto añadiendo un método al expositor que nos deje devolver un trozo de tarta. Tendremos que añadir manejo de excepciones para garantizar que no nos quedamos con recursos si algo falla:

```
slice = display_case.get_pie_if_available()
if slice
   try {
      scoop = freezer.get_ice_cream_if_available()
      if scoop
      try {
         give_order_to_customer()
      }
rescue {
         freezer.give_back(scoop)
      }
      end
   }
   rescue {
   display_case.give_back(slice)
   }
end
```

De nuevo, esto no es lo ideal. Ahora el código es muy feo, determinar qué hace de verdad es difícil: la lógica de negocio está enterrada bajo todo el mantenimiento.

Antes hemos arreglado esto moviendo el código de gestión del recurso dentro del propio recurso. Aquí, sin embargo, tenemos dos recursos. ¿Deberíamos poner el código en el expositor o en el congelador?

Creemos que la respuesta es “no” para ambas opciones. El enfoque pragmático sería decir que “tarta de manzana con bola de helado” es su propio recurso. Moveríamos este código a un módulo nuevo y, después, el cliente podría decir solo “tráigame tarta de manzana con helado” y, bien tiene éxito, bien falla.

Por supuesto, en el mundo real es probable que haya muchos platos como este, y no querríamos escribir módulos nuevos para cada uno. En vez de eso, seguramente nos interesaría tener algún tipo de elemento de menú que contuviese referencias a sus componentes y, después, tener un método `get_menu_item` genérico que hiciese el baile de los recursos con cada uno.

Actualizaciones no transaccionales

Se presta mucha atención a la memoria compartida como fuente de problemas de concurrencia, pero, en realidad, los problemas pueden surgir en cualquier parte donde el código de la aplicación comparta recursos mutables: archivos, bases de datos, servicios externos, etc. Cada vez que dos o más instancias del código pueden acceder al mismo recurso al mismo tiempo, podemos ver un problema potencial.

A veces, el recurso no es tan obvio. Mientras escribíamos esta edición del libro, actualizamos la cadena de herramientas para hacer más trabajo en paralelo usando hilos. Esto hizo que la construcción fallase, pero de maneras extrañas y en lugares aleatorios. Un problema común a todos los errores era que había archivos o directorios que no podían encontrarse, incluso aunque en realidad estaban justo en el lugar correcto.

Rastreamos esto hasta un par de lugares del código que cambiaban de manera temporal el directorio actual. En la versión no paralela, el hecho de que este código restaurase otra vez el directorio era suficiente, pero, en la versión paralela, un hilo cambiaba el directorio y después, mientras estaba en ese directorio, otro hilo empezaba a ejecutarse. Ese hilo esperaba estar en el directorio original, pero, como el directorio actual se comparte entre hilos, no era el caso.

La naturaleza de este problema genera otro truco:

Truco 58. Los fallos aleatorios son a menudo problemas de concurrencia.

Otros tipos de accesos exclusivos

La mayoría de lenguajes tienen soporte de biblioteca para algún tipo de acceso a recursos compartidos. Puede que los llamen *mutex* (de *mutual exclusion*, exclusión mutua), monitores o semáforos. Todos se implementan como bibliotecas.

Sin embargo, algunos lenguajes tienen soporte de concurrencia integrado en el propio lenguaje. Rust, por ejemplo, impone el concepto de la propiedad de los datos; solo una variable o parámetro puede albergar una referencia a cualquier dato mutable concreto a la vez.

También podría argumentarse que los lenguajes funcionales, con su tendencia a hacer todos los datos inmutables, hacen que la concurrencia sea más simple. Sin embargo, siguen enfrentándose a los mismos retos, porque, en algún momento, se ven obligados a entrar en el mundo real mutable.

Doctor, me hago daño...

Incluso aunque no aproveche nada más de esta sección, quédese con esto: la concurrencia en un entorno de recursos compartidos es difícil y gestionarla usted mismo supone múltiples desafíos.

Por eso le recomendamos el remate de un viejo chiste:

—Doctor, me duele al hacer esto.
—Entonces no lo haga.

El siguiente par de secciones sugieren maneras alternativas de obtener los beneficios de la concurrencia, pero sin el dolor.

Las secciones relacionadas incluyen

- Tema 10, “Ortogonalidad”.
- Tema 28, “Desacoplamiento”.
- Tema 38, “Programar por casualidad”.

35 Actores y procesos

Sin escritores, la historias no se escribirían; sin actores, las historias no cobrarían vida.

—Angie-Marie Delsante

Los actores y los procesos ofrecen maneras interesantes de implementar la concurrencia sin la carga de tener que sincronizar el acceso a la memoria compartida.

Antes de hablar de ellas, sin embargo, tenemos que definir qué queremos decir. Esto va a sonar académico. No se preocupe, en poco tiempo lo revisaremos todo.

- Un actor es un procesador virtual independiente con su propio estado local (y privado). Cada actor tiene un buzón. Cuando aparece un mensaje en el buzón y el actor está desocupado, se pone en marcha y procesa el mensaje. Cuando termina con el procesamiento, pasa a procesar otro mensaje del buzón o, si el buzón está vacío, vuelve a dormirse. Cuando está procesando un mensaje, un actor puede crear otros actores, enviar mensajes a otros actores cuya existencia conozca y crear un estado nuevo que se convertirá en el estado actual cuando se procese el siguiente mensaje.
- Un procesador suele ser un procesador virtual para fines más generales, a menudo implementado por el sistema operativo para facilitar la concurrencia. Los procesos pueden estar restringidos (por convención) para comportarse como actores, y ese es el tipo de proceso al que nos referimos aquí.

Los actores solo pueden ser concurrentes

Hay algunas cosas que no encontrará en la definición de los actores:

- No hay una sola cosa que tenga el control. Nada programa lo que va a pasar a continuación ni orquesta la transferencia de información de los datos brutos a la salida final.
- El único estado en el sistema está albergado en mensajes y en el estado local de cada actor. Los mensajes no pueden ser examinados

más que mediante su lectura por parte de su destinatario y el estado local es inaccesible fuera del actor.

- Todos los mensajes van en una dirección; no existe el concepto de responder. Si quiere que un actor devuelva una respuesta, incluye la dirección de su propio buzón en el mensaje que le envía, y, al final, el actor enviará la respuesta como cualquier otro mensaje a ese buzón.
- Un actor procesa cada mensaje hasta su completitud y solo procesa los mensajes de uno en uno.

Como resultado, los actores se ejecutan de manera concurrente y asíncrona y no comparten nada. Si tuviese suficientes procesadores físicos, podría ejecutar un actor en cada uno. Si tiene un solo procesador, entonces algo de tiempo de ejecución puede gestionar el cambio de contexto entre ellos. En cualquier caso, el código que se ejecuta en los actores es el mismo.

Truco 59. Use actores para lograr concurrencia sin estado compartido.

Un actor simple

Vamos a implementar nuestro restaurante utilizando actores. En este caso, tendremos tres (el cliente, el camarero y el expositor de la tarta).

El flujo del mensaje general tendrá este aspecto:

- Nosotros (como algún tipo de ser divino) le decimos al cliente que tiene hambre.
- En respuesta, le pide tarta al camarero.
- El camarero pedirá al expositor de la tarta un trozo para el cliente.
- Si el expositor de la tarta tiene un trozo disponible, lo enviará al cliente y también notificará al camarero que debe añadirlo a la cuenta.
- Si no hay tarta, el expositor se lo dice al camarero y este se disculpa con el cliente.

Hemos elegido implementar el código en JavaScript usando la biblioteca Nact.[3] Hemos añadido un pequeño envoltorio a esto que nos permite

escribir actores como objetos simples, donde las claves son los tipos de mensaje que recibe y los valores son funciones que se ejecutan cuando se recibe ese mensaje en particular. (La mayoría de los sistemas de actores tienen un tipo de estructura similar, pero los detalles dependen del lenguaje anfitrión).

Vamos a empezar por el cliente. El cliente puede recibir tres mensajes:

- Tiene hambre (enviado por el contexto externo).
- Hay tarta en el menú (enviado por el expositor de la tarta).
- Lo siento, no hay tarta (enviado por el camarero).

Este es el código:

concurrency/actors/index.js

```
const customerActor = {
  'hungry for pie': (msg, ctx, state) => {
    return dispatch(state.waiter,
              { type: "order", customer: ctx.self, wants: 'pie' })
  },

  'put on table': (msg, ctx, _state) =>
    console.log(`${ctx.self.name} sees "${msg.food}" appear on the
table`),

  'no pie left': (_msg, ctx, _state) =>
    console.log(`${ctx.self.name} sulks...`)
}
```

El caso interesante se da cuando recibimos un mensaje "hambre de tarta", donde enviamos un mensaje al camarero. (Enseguida veremos cómo conoce el cliente al actor camarero).

Veamos el código del camarero:

concurrency/actors/index.js

```
const waiterActor = {
  "order": (msg, ctx, state) => {
    if (msg.wants == "pie") {
      dispatch(state.pieCase,
           { type: "get slice", customer: msg.customer, waiter:
ctx.self })
```

```
  }
  else {
    console.dir(`Don't know how to order ${msg.wants}`);
  }
  },

  "add to order": (msg, ctx) =>
    console.log(`Waiter adds ${msg.food} to ${msg.customer.name}'s
order`),

  "error": (msg, ctx) => {
    dispatch(msg.customer, { type: 'no pie left', msg: msg.msg });
    console.log(`\nThe waiter apologizes to ${msg.customer.name}:
${msg.msg}`)
  }

};
```

Cuando recibe el mensaje `'order'` (pedido) del cliente, comprueba si la solicitud es de tarta. Si es así, envía una solicitud al expositor de la tarta, pasando referencias tanto a sí mismo como al cliente.

El expositor de la tarta tiene estado: una matriz de todos los trozos de tarta que alberga. (De nuevo, veremos cómo se configura eso enseguida). Cuando recibe un mensaje `'get slice'` (obtener trozo), ve si queda algún trozo. Si es así, pasa el trozo al cliente, le dice al camarero que actualice el pedido y, por último, devuelve un estado actualizado que contiene un trozo menos. Este es el código:

concurrency/actors/index.js

```
const pieCaseActor = {
  'get slice': (msg, context, state) => {
    if (state.slices.length == 0) {
      dispatch(msg.waiter,
           { type: 'error', msg: "no pie left", customer:
msg.customer })
      return state
  }
  else {
      var slice = state.slices.shift() + " pie slice";
      dispatch(msg.customer,
           { type: 'put on table', food: slice });
      dispatch(msg.waiter,
           { type: 'add to order', food: slice, customer:
msg.customer });
```

```
        return state;
      }
    }
}
```

Aunque a menudo descubrirá que los actores se inician de forma dinámica por otros, en nuestro caso vamos a mantenerlo simple y a iniciar los actores de manera manual. También pasaremos a cada uno un estado inicial:

- El expositor de la tarta obtiene la lista inicial de trozos de tarta que contiene.
- Daremos al camarero una referencia al expositor de la tarta.
- Daremos a los clientes una referencia al camarero.

concurrency/actors/index.js

```
const actorSystem = start();

let pieCase = start_actor(
   actorSystem,
   'pie-case',
   pieCaseActor,
   { slices: ["apple", "peach", "cherry"] });

let waiter = start_actor(
   actorSystem,
   'waiter',
   waiterActor,
   { pieCase: pieCase });
let c1 = start_actor(actorSystem, 'customer1',
             customerActor, { waiter: waiter });
let c2 = start_actor(actorSystem, 'customer2',
             customerActor, { waiter: waiter });
```

Y, por último, lo ponemos en marcha. Nuestros clientes son glotones. El cliente 1 pide tres trozos de tarta y el cliente 2 pide dos:

concurrency/actors/index.js

```
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
dispatch(c2, { type: 'hungry for pie', waiter: waiter });
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
```

```
dispatch(c2, { type: ‘hungry for pie’, waiter: waiter });
dispatch(c1, { type: ‘hungry for pie’, waiter: waiter });
sleep(500)
     .then(() => {
     stop(actorSystem);
  })
```

Cuando lo ejecutamos, podemos ver a los actores comunicándose.[4] El orden que ve usted puede ser diferente:

```
$ node index.js
el cliente1 ve que “trozo de tarta de manzana” aparece en el menú
el cliente2 ve que “trozo de tarta de melocotón” aparece en el menú
El camarero añade trozo de tarta de manzana al pedido del cliente1
El camarero añade trozo de tarta de melocotón al pedido del
cliente2
el cliente1 ve que “trozo de tarta de cereza” aparece en el menú
El camarero añade trozo de tarta de cereza al pedido del cliente1

El camarero se disculpa con el cliente1: no queda tarta
el cliente1 se enfurruña...

El camarero se disculpa con el cliente2: no queda tarta
el cliente2 se enfurruña...
```

Sin concurrencia explícita

En el modelo actor, no hay necesidad de escribir ningún código para manejar la concurrencia, puesto que no hay estado compartido. Tampoco es necesario escribir código con una lógica explícita “haz esto, haz aquello” de extremo a extremo, ya que los actores lo averiguan por sí mismos basándose en los mensajes que reciben.

Tampoco hay mención de la arquitectura subyacente. Este conjunto de componentes funciona igual de bien en un solo procesador, en múltiples núcleos o en múltiples máquinas conectadas en red.

Erlang prepara el escenario

El lenguaje y el tiempo de ejecución Erlang son ejemplos estupendos de una implementación de actores (incluso aunque los creadores de Erlang no habían leído el artículo original sobre actores). Erlang llama a los actores

“procesos”, pero no son los procesos del sistema operativo corrientes, sino que, como los actores de los que hemos estado hablando, los procesos de Erlang son ligeros (podemos ejecutar millones de ellos en una sola máquina) y se comunican enviando mensajes. Cada uno está aislado de los demás, así que no se comparte el estado.

Además, el tiempo de ejecución de Erlang implementa un sistema de supervisión, que gestiona la vida de los procesos, reiniciando potencialmente un proceso o conjunto de procesos en caso de que se produzca un fallo. Y Erlang también ofrece carga de código en caliente: podemos reemplazar código en un sistema en ejecución sin detener dicho sistema. Y el sistema Erlang ejecuta algunos de los códigos más fiables del mundo, citando a menudo una disponibilidad de nueve nueves. Pero Erlang (y su progenie Elixir) no son únicos; hay implementación de actores para la mayoría de los lenguajes. Plantéese utilizarlos para sus implementaciones concurrentes.

Las secciones relacionadas incluyen

- Tema 28, “Desacoplamiento”.
- Tema 30, “Transformar la programación”.
- Tema 36, “Pizarras”.

Retos

- ¿Tiene en este momento código que utiliza la exclusión mutua para proteger datos compartidos? ¿Por qué no prueba un prototipo del mismo código escrito utilizando actores?
- El código actor para el restaurante solo soporta pedidos de trozos de tarta. Amplíelo para permitir que los clientes pidan tarta con helado, con agentes separados gestionando los trozos de tarta y las bolas de helado. Organícelo todo de manera que se pueda manejar la situación en la que se acaba una de las dos cosas.

36 Pizarras

La escritura en la pared...

—Daniel 5 (referencia)

Piense en cómo los detectives utilizarían una pizarra para coordinarse y resolver un asesinato. La inspectora jefe empieza por llevar una pizarra grande a la sala de conferencias. En ella, escribe una sola pregunta:

H. Dumpty (Varón, Huevo): ¿Accidente? ¿Asesinato?

¿Humpty se cayó de verdad o fue empujado? Cada detective puede contribuir a este misterio de asesinato potencial añadiendo hechos, declaraciones de testigos, pruebas forenses que puedan surgir, etc. A medida que se acumulan datos, puede que un detective se dé cuenta de que hay una conexión y escriba también esa observación o especulación. Este proceso continúa a lo largo de todos los turnos, con muchas personas y agentes diferentes, hasta que se cierra el caso. La figura 6.3 muestra una pizarra de ejemplo.

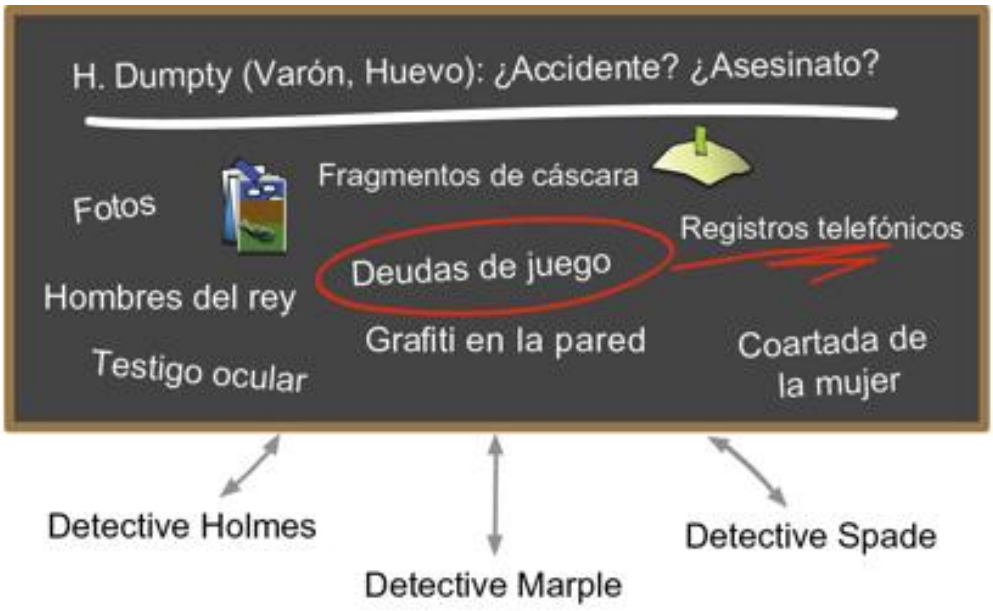


Figura 6.3. Alguien descubrió una conexión entre las deudas de juego de Humpty y los registros telefónicos. Quizá le amenazaban por teléfono.

Algunas características clave del enfoque de las pizarras son:

- Ninguno de los detectives necesita conocer la existencia de ningún otro detective; miran la pizarra para ver la información nueva y

añaden sus hallazgos.

- Los detectives pueden tener formación en distintas disciplinas, pueden tener diferentes niveles de educación y experiencia distintos y tal vez ni siquiera trabajen en la misma comisaría. Comparten el deseo de resolver el caso, pero eso es todo.
- Diferentes detectives pueden ir y venir en el transcurso del proceso y pueden trabajar en turnos distintos.
- No hay restricciones en cuanto a lo que puede ponerse en la pizarra. Pueden ser fotos, frases, pruebas físicas, etc.

Se trata de una forma de concurrencia de "dejar hacer". Los detectives son procesos, agentes, actores, etc., independientes. Algunos almacenan hechos en la pizarra. Otros sacan hechos de la pizarra, quizá combinándolos o procesándolos, y añaden más información a esta. De manera gradual, la pizarra les ayuda a llegar a una conclusión.

Los sistemas de pizarras basados en ordenador se utilizaban en sus inicios en aplicaciones de inteligencia artificial donde los problemas que había que resolver eran grandes y complejos: reconocimiento de discurso, sistemas de razonamiento basados en conocimiento, etc.

Uno de los primeros sistemas de pizarra fue Linda, de David Gelernter. Almacenaba hechos como tuplas tipadas. Las aplicaciones podían escribir tuplas nuevas en Linda y consultar las existentes mediante una forma de coincidencia de patrones.

Más tarde, llegaron los sistemas distribuidos similares a las pizarras, como JavaSpaces y T Spaces. Con estos sistemas, podemos almacenar objetos Java activos (no solo datos) en la pizarra y recuperarlos mediante una coincidencia parcial de los campos (a través de plantillas o comodines) o mediante subtipos. Por ejemplo, supongamos que tenemos un tipo `Autor`, que es un subtipo de `Persona`. Podríamos hacer una búsqueda en una pizarra que contenga objetos `Persona` utilizando una plantilla `Autor` con un valor `apellido` de "Shakespeare". Obtendríamos Bill Shakespeare el autor, pero no Fred Shakespeare el jardinero.

Creemos que estos sistemas nunca llegaron a despegar del todo en parte porque la necesidad del tipo de procesamiento cooperativo concurrente todavía no se había desarrollado.

Una pizarra en acción

Supongamos que estamos escribiendo un programa para aceptar y procesar solicitudes de hipotecas o préstamos. Las leyes que rigen esta área son complejas de una manera odiosa, con los gobiernos federal, estatal y local metiéndose en el asunto. El prestamista debe probar que ha revelado determinadas cosas y debe pedir cierta información, pero no debe hacer otras preguntas determinadas, etc.

Más allá del miasma de leyes aplicables, también tenemos los siguientes problemas de los que ocuparnos:

- Las respuestas pueden llegar en cualquier orden. Por ejemplo, las consultas para una comprobación de crédito o búsqueda de títulos de propiedades pueden llevar un tiempo considerable, mientras que elementos como el nombre y la dirección pueden estar disponibles al instante.
- La recopilación de datos pueden llevarla a cabo personas diferentes, distribuidas por oficinas distintas, en diferentes zonas horarias.
- Algunas recopilaciones de datos pueden hacerlas de forma automática otros sistemas. Estos datos también pueden llegar de manera asíncrona.
- Sin embargo, puede que algunos datos todavía dependan de otros datos. Por ejemplo, puede que no podamos empezar la búsqueda de la titularidad de un coche hasta que no recibamos la prueba de la propiedad o el seguro.
- La llegada de datos nuevos puede plantear nuevas preguntas y políticas. Supongamos que la comprobación de crédito genera un informe poco brillante; ahora hacen falta estos cinco formularios extra y, quizá, una muestra de sangre.

Podemos intentar gestionar cualquier combinación y circunstancia posible usando un sistema de flujo de trabajo. Existen muchos sistemas así, pero pueden ser complejos y requerir mucha programación. A medida que cambian las regulaciones, hay que reorganizar el flujo de trabajo: puede que la gente tenga que cambiar sus procedimientos y que haya que reescribir el código intrínseco.

Una pizarra, en combinación con un motor de reglas que encapsule los requisitos legales, es una solución elegante para las dificultades que encontramos aquí. El orden de llegada de los datos es irrelevante: cuando se publica un hecho, puede desencadenar las reglas apropiadas. El *feedback* también se gestiona con facilidad: la salida de cualquier conjunto de reglas puede publicarse en la pizarra y provocar la activación de más reglas aplicables aún.

Truco 60. Use pizarras para coordinar el flujo de trabajo.

Los sistemas de mensajería pueden ser como pizarras

Mientras escribimos esta segunda edición, están construyéndose muchas aplicaciones utilizando servicios pequeños y desacoplados que se comunican a través de algún tipo de sistema de mensajería. Estos sistemas de mensajería (como Kafka y NATS) hacen mucho más que enviar datos de A a B sin más. En particular, ofrecen persistencia (en forma de registro de eventos) y la capacidad para recuperar mensajes a través de una forma de coincidencia de patrones. Esto significa que podemos usarlos como sistema de pizarra y también como plataforma en la que podemos ejecutar un puñado de actores.

Pero no es tan simple...

El enfoque de los actores o las pizarras o los microservicios respecto a la arquitectura elimina una clase entera de problemas potenciales de concurrencia de nuestras aplicaciones, pero ese beneficio tiene un precio. Es más difícil razonar acerca de estos enfoques, porque gran parte de la acción es indirecta. Descubrirá que resulta útil tener un repositorio central de API o formatos de mensajes, sobre todo si ese repositorio puede generar el código y la documentación por usted. También necesitará buenas herramientas para poder rastrear mensajes y hechos a medida que avancen por el sistema. (Una técnica útil es añadir un identificador de rastreo único cuando se inicie una función de negocio determinada y, después, propagarlo

a todos los actores involucrados. Entonces, será capaz de reconstruir lo que sucede desde los archivos de registro).

Por último, estos tipos de sistemas pueden ser más problemáticos a la hora de desplegarlos y gestionarlos, ya que hay más partes móviles. Hasta cierto punto, esto se compensa con el hecho de que el sistema es más granular y puede actualizarse mediante la sustitución de actores individuales y no de todo el sistema.

Las secciones relacionadas incluyen

- Tema 28, “Desacoplamiento”.
- Tema 29, “Malabares con el mundo real”.
- Tema 33, “Romper el acoplamiento temporal”.
- Tema 35, “Actores y procesos”.

Ejercicios

Ejercicio 24

¿Sería apropiado un sistema de estilo pizarra para las siguientes aplicaciones? ¿Por qué o por qué no?

Procesamiento de imágenes. Le gustaría hacer que varios procesos paralelos tomasen porciones de una imagen, los procesasen y devolviesen la porción terminada.

Agenda para grupos. Tiene a varias personas diseminadas por todo el mundo, en diferentes zonas horarias y que hablan idiomas distintos, intentando programar una reunión.

Herramienta de monitorización de red. El sistema recopila estadísticas de rendimiento y recoge informes de problemas, que los agentes utilizan para buscar problemas en el sistema.

Retos

- ¿Utiliza sistemas de pizarras en el mundo real, un tablón con mensajes al lado del frigorífico o una pizarra blanca grande en el

trabajo? ¿Qué hace que sean efectivos? ¿Se publican los mensajes con formatos consistentes? ¿Importa eso?

---

[1] Aunque el UML ha ido desapareciendo de manera gradual, muchos de sus diagramas individuales siguen existiendo de una forma u otra, incluyendo el utilísimo diagrama de actividad. Para obtener más información acerca de todos los tipos de diagrama UML, consulte *UML Distilled: A Brief Guide to the Standard Object Modeling Language* [Fow04].

[2] Los nombres P y V vienen de las iniciales de palabras neerlandesas. Sin embargo, hay cierto debate acerca de qué palabras son. El inventor de la técnica, Edsger Dijkstra, sugirió tanto *passering* como *prolaag* para P, y *vrijgave* y, posiblemente, *verhogen* para V.

[3] https://github.com/ncthbrt/nact.

[4] Para ejecutar este código, también necesitará nuestras funciones de envoltorio, que no se muestran aquí. Puede descargarlas en https://media.pragprog.com/titles/tpp20/code/concurrency/actors/index.js.

7
Mientras escribe código

La sabiduría popular dice que, una vez que un proyecto está en la fase de escritura del código, el trabajo es sobre todo mecánico, para transcribir el diseño como sentencias ejecutables. Nosotros creemos que esta actitud es la principal razón por la que fracasan los proyectos de software y muchos sistemas acaban siendo feos, ineficientes, con una estructura pobre, imposibles de mantener o, simplemente, que están mal.

La escritura del código no es mecánica. Si lo fuese, las herramientas CASE en las que la gente depositó sus esperanzas a principios de los ochenta habrían reemplazado a los programadores hace mucho tiempo. Hay que tomar decisiones a cada minuto, decisiones que requieren una reflexión, cuidados y un buen criterio si se quiere que el programa resultante disfrute de una vida larga, productiva y precisa.

No todas las decisiones son conscientes siquiera. Para aprovechar mejor sus instintos y pensamientos no conscientes, “Escuche a su cerebro reptiliano”. Veremos cómo escuchar con más atención y buscar maneras de responder de forma activa a esos pensamientos que en ocasiones son exasperantes.

Pero escuchar a sus instintos no significa que pueda volar con el piloto automático puesto. Los desarrolladores que no piensan de forma activa en su código están programando por casualidad; quizá el código funcione, pero no hay ninguna razón en particular para que lo haga. En “Programar por casualidad”, abogamos por una implicación más positiva en el proceso de escritura del código.

Aunque la mayor parte del código que escribimos se ejecuta con rapidez, de vez en cuando desarrollamos algoritmos que tienen el potencial para atascar incluso los procesadores más veloces. En “Velocidad de los algoritmos”, veremos maneras de calcular la velocidad del código y ofreceremos algunos trucos para detectar problemas potenciales antes de que ocurran.

Los programadores pragmáticos miramos con una visión crítica todo el código, incluido el nuestro. Vemos todo el tiempo espacio para la mejora de nuestros programas y diseños.

En “Refactorización”, veremos técnicas que nos ayudan a arreglar el código existente de manera continua a medida que avanzamos.

Las pruebas no son para encontrar fallos, son para recibir *feedback* sobre nuestro código: aspectos de diseño, la API, el acoplamiento, etc. Eso significa que los beneficios principales de las pruebas se dan cuando pensamos en ellas y las escribimos, no solo cuando las ejecutamos. Exploraremos esta idea en “Probar para escribir código”.

Pero, por supuesto, cuando probamos nuestro propio código, puede que introduzcamos nuestros propios sesgos en la tarea. En “Pruebas basadas en propiedades” veremos cómo hacer que el ordenador realice pruebas de amplio espectro por nosotros y cómo manejar los fallos inevitables que surgirán.

Es crucial que escribamos código que sea legible y sobre el que sea fácil razonar. El mundo exterior es duro, está lleno de actores malos que intentarán de forma activa irrumpir en nuestro sistema y causar daños. Hablaremos de técnicas y enfoques muy básicos para ayudar a que “Tenga cuidado ahí fuera”.

Por último, uno de los aspectos más difíciles del desarrollo de software es “Poner nombre a las cosas”. Tenemos que poner nombre a muchas cosas y, en muchos sentidos, los nombres que elegimos definen la realidad que creamos. Necesitamos estar atentos a cualquier deriva semántica potencial mientras escribimos código.

La mayoría de nosotros conducimos el coche sobre todo con el piloto automático puesto; no ordenamos de forma explícita a nuestro pie que pise un pedal ni a nuestro brazo que gire el volante; solo pensamos “reducir la velocidad y girar a la derecha”. Sin embargo, los buenos conductores que se preocupan por la seguridad están revisando la situación todo el tiempo, buscando problemas potenciales y colocándose en buenas posiciones por si ocurre algo imprevisto. Lo mismo se aplica a la escritura del código; puede que sea en gran parte una rutina, pero mantener la sensatez podría evitar un desastre.

37 Escuche a su cerebro reptiliano

Solo los seres humanos pueden mirar directamente algo, tener toda la información que necesitan para hacer una predicción precisa, quizá incluso hacer momentáneamente la predicción precisa, y después decir que no es así.

—Gavin de Becker, El valor del miedo

El trabajo de la vida de Gavin de Becker es ayudar a las personas a protegerse a sí mismas. Su libro *El valor del miedo: señales de alarma que nos protegen de la violencia* [de 98] encapsula este mensaje. Uno de los temas clave que se tratan a lo largo del libro es que, como humanos sofisticados, hemos aprendido a ignorar nuestro lado más animal; nuestros instintos, nuestro cerebro reptiliano. Afirma que la mayoría de las personas a las que atacan en la calle son conscientes de que se sienten incómodas o nerviosas antes del ataque, pero se dicen a sí mismas que están siendo tontas. Entonces, la figura sale del portal oscuro...

Los instintos son simplemente una respuesta a patrones que se encuentran en nuestro cerebro inconsciente. Algunos son innatos y otros se aprenden mediante la repetición. A medida que ganamos experiencia como programadores, nuestro cerebro va acumulando capas de conocimiento tácito: cosas que funcionan, cosas que no funcionan, las causas probables de un tipo de error, todas las cosas de las que nos damos cuenta a lo largo del día. Esta es la parte de nuestro cerebro que pulsa la tecla para guardar un archivo cuando hacemos una pausa para hablar con alguien, incluso aunque no nos demos cuenta de que lo estamos haciendo.

Sea cual sea su fuente, los instintos comparten una cosa: no tienen palabras. Los instintos nos hacen sentir, no pensar. Por tanto, cuando un instinto se activa, no vemos cómo se enciende una bombilla con una pancarta alrededor, sino que nos ponemos nerviosos, nos sentimos revueltos o nos parece que esto es demasiado trabajo.

El truco está en notar primero lo que está pasando y, después, determinar el porqué. Vamos a empezar por echar un vistazo a un par de situaciones comunes en las que nuestro reptil interior está intentando decirnos algo. Después, veremos cómo podemos sacar ese cerebro instintivo de su envoltorio protector.

Miedo a la página en blanco

Todo el mundo teme a la pantalla vacía, al solitario cursor parpadeante rodeado de un montón de nada. Empezar un proyecto nuevo (o incluso un módulo nuevo en un proyecto existente) puede ser una experiencia enervante. Muchos de nosotros preferiríamos aplazar el compromiso inicial de empezar.

Creemos que hay dos problemas que causan esto, y que ambos tienen la misma solución. Un problema es que su cerebro reptiliano está intentando decirle algo; hay algún tipo de duda acechándole justo debajo de la superficie de percepción, y eso es importante. Como desarrollador, ha estado probando cosas y viendo qué funcionaba y qué no. Ha estado acumulando experiencia y sabiduría. Cuando sienta una duda molesta o experimente cierta reticencia cuando se enfrente a una tarea, quizá sea su experiencia intentando decirle algo. Préstele atención. Puede que no sea capaz de señalar con exactitud lo que va mal, pero dé algo de tiempo y es probable que sus dudas acaben cristalizando en algo más sólido, algo que pueda abordar. Deje que sus instintos contribuyan a su rendimiento.

El otro problema es un poco más prosaico: simplemente tiene miedo de cometer un error. Y es un temor razonable. Los desarrolladores ponemos mucho de nosotros mismos en nuestro código; podemos tomarnos los errores en ese código como un reflejo de nuestra competencia. Quizá haya también un elemento de síndrome del impostor; puede que pensemos que este proyecto es demasiado para nosotros. No podemos ver la forma de llegar hasta el final; llegaremos hasta un punto concreto y, después, nos veremos obligados a admitir que estamos perdidos.

Luche contra sí mismo

A veces, el código vuela desde el cerebro al editor: las ideas se convierten en bits sin esfuerzo aparente.

Otros días, parece que el código va subiendo por una colina llena de barro. Cada paso requiere un esfuerzo tremendo y cada tres pasos retrocedemos dos. Pero, como es profesional, no afloja, sigue dando paso tras paso en el barro: tiene un trabajo que hacer. Por desgracia, es probable que eso sea justo lo contrario de lo que debería hacer.

Su código está intentando decirle algo. Le está diciendo que esto es más difícil de lo que debería ser. Quizá la estructura o el diseño estén mal, tal vez está resolviendo el problema equivocado o puede que esté creando un millón de fallos. Sea cual sea el motivo, su cerebro reptiliano está sintiendo el *feedback* que envía el código y está intentando desesperadamente que lo escuche.

Cómo hablar lagarto

Hablamos mucho acerca de escuchar a nuestros instintos, a nuestro cerebro reptiliano inconsciente. Las técnicas son siempre las mismas.

Truco 61. Escuche a su cerebro reptiliano.

Primero, deje lo que esté haciendo. Dese algo de tiempo y espacio para permitir que su cerebro se organice. Deje de pensar en el código y haga algo que no requiera mucha concentración, lejos del teclado. Dé un paseo, coma, charle con alguien. Consúltelo con la almohada. Deje que las ideas se filtren a través de las capas de su cerebro por sí solas: no puede forzarlo. Al final, puede que suban hasta el nivel consciente y tenga uno de esos momentos de "¡eureka!".

Si eso no funciona, pruebe a externalizar el problema. Haga garabatos sobre el código que está escribiendo o explíqueselo a un compañero (a poder ser, alguien que no sea programador) o a su patito de goma. Exponga diferentes partes de su cerebro al problema y vea si alguna de ellas tiene un entendimiento mejor de lo que esté preocupándole. Hemos perdido la cuenta de la cantidad de conversaciones que hemos tenido en las que uno de nosotros explicaba un problema al otro y de repente decía: "¡Ah! ¡Claro!" y salía corriendo a arreglarlo.

Pero puede que haya probado estas cosas y siga atascado. Ha llegado la hora de la acción. Tenemos que decirle al cerebro que lo que está a punto de hacer no importa, y eso se hace mediante la creación de prototipos.

¡Hora de jugar!

Andy y Dave han pasado horas mirando búferes vacíos del editor. Escribimos algo de código, miramos al techo, vamos a por otra bebida, después tecleamos algo más de código, luego leemos una historia graciosa sobre un gato con dos rabos, escribimos algo más de código, después seleccionamos todo y eliminamos y empezamos otra vez. Y otra vez. Y otra.

Y con el paso de los años hemos descubierto un truco cerebral que parece funcionar. Dígase a sí mismo que necesita crear un prototipo de algo. Si está enfrentándose a una pantalla en blanco, busque algún aspecto del proyecto que quiera explorar. A lo mejor está utilizando un *framework* nuevo y quiere ver cómo hace la vinculación de datos. O quizá es un nuevo algoritmo y quiere explorar cómo funciona en casos extremos. O puede que quiera probar un par de estilos diferentes de interacción de usuarios.

Si está trabajando con código existente y este está oponiendo resistencia, escóndalo en otra parte y cree en su lugar un prototipo de algo similar. Haga lo siguiente.

1. Escriba “Estoy haciendo un prototipo” en una nota adhesiva y péguela en un lado de la pantalla.
2. Recuérdese que los prototipos están pensados para fallar. Y recuérdese que los prototipos se desechan, incluso aunque no fallen. No hay ningún inconveniente en esto.
3. En el búfer vacío del editor, cree un comentario que describa en una frase lo que quiere aprender o hacer.
4. Empiece a escribir código.

Si comienza a tener dudas, mire la nota adhesiva.

Si mientras está escribiendo código esa duda molesta se cristaliza de pronto y se convierte en una preocupación sólida, abórdela.

Si llega al final del experimento y todavía siente cierta incomodidad, empiece de nuevo con el paseo y la charla y el descanso.

Pero, según nuestra experiencia, en algún momento durante la creación del primer prototipo se sorprenderá al encontrarse tarareando al ritmo de su música, disfrutando el sentimiento de crear código. El nerviosismo se evaporará y dejará paso a una sensación de urgencia: ¡acabemos esto!

En esta etapa, sabe lo que tiene que hacer. Elimine todo el código del prototipo, tire la nota adhesiva y llene el búfer vacío del editor con código nuevo y brillante.

No solo su código

Una gran parte de su trabajo consiste en tratar con código existente, a menudo escrito por otras personas. Esas personas tendrán instintos diferentes de los suyos, así que las decisiones que tomen serán distintas. No tienen por qué ser peores; solo diferentes.

Puede leer su código de forma mecánica, mirándolo a fondo y tomando notas de aquello que parezca importante. Es una lata, pero funciona.

O puede probar a hacer un experimento. Cuando detecte cosas hechas de una manera que parezca extraña, anótelo. Siga haciéndolo y busque patrones. Si puede ver lo que llevó a esa persona a escribir el código de ese modo, puede que la tarea de entenderlo resulte mucho más fácil. Será capaz de aplicar de manera consciente los patrones que ella aplicó de forma tácita.

Y puede que aprenda algo por el camino.

No solo código

Aprender a seguir su instinto al escribir código es una habilidad importante que fomentar, pero se aplica también a aspectos más generales. A veces, un diseño da malas sensaciones o algún requisito produce incomodidad. Pare y analice estos sentimientos. Si está en un entorno que le apoye, expréselos en voz alta. Explórelos. Hay posibilidades de que haya algo acechando en el portal oscuro. Escuche a sus instintos y evite el problema antes de que se lance sobre usted.

Las secciones relacionadas incluyen

- Tema 13, “Prototipos y notas en *post-its*”.
- Tema 22, “Cuadernos de bitácora de ingeniería”.
- Tema 46, “Resolver rompecabezas imposibles”.

Retos

- ¿Hay algo que sepa que debería hacer, pero que haya estado posponiendo porque le parece difícil o le da un poco de miedo? Aplique las técnicas de esta sección. Póngase un temporizador de una hora, quizá dos, y prométase que cuando suene la campana eliminará lo que ha hecho. ¿Qué ha aprendido?

38 Programar por casualidad

¿Alguna vez ve viejas películas bélicas en blanco y negro? El soldado agotado sale con cautela de la maleza. Hay un claro ante él: ¿está lleno de minas o es seguro cruzarlo? No hay nada que indique que es un campo de minas; no hay señales, ni alambradas ni cráteres. El soldado toca el suelo delante de él con su bayoneta y se encoge, esperando una explosión. No hay ninguna. Así que sigue avanzando con cuidado por el campo durante un rato, tocando y pinchando a medida que se mueve. Al final, convencido de que el terreno es seguro, se yergue y camina hacia delante con orgullo, hasta que vuela en pedazos.

El sondeo inicial del soldado en busca de minas no había revelado nada, pero había sido solo suerte. Había llegado a una conclusión falsa, con consecuencias desastrosas.

Como desarrolladores, también trabajamos en campos de minas. Hay cientos de trampas esperando para pillarnos cada día. Recordando la historia del soldado, deberíamos ser cautos a la hora de sacar falsas conclusiones. Deberíamos evitar programar por casualidad (depender de la suerte y los éxitos accidentales) y, en vez de eso, programar de forma deliberada.

Cómo programar por casualidad

Supongamos que Fred recibe un encargo de programación. Fred escribe algo de código, lo prueba y parece que funciona. Escribe un poco más, lo prueba y aún parece que funciona. Después de varias semanas de escribir código de este modo, el programa deja de funcionar de repente y, después

de varias horas intentando arreglarlo, Fred todavía no sabe por qué. Puede pasar una cantidad importante de tiempo dando vueltas a este código sin ser capaz de arreglarlo. Da igual lo que haga, parece que nada funciona como debería.

Fred no sabe por qué el código está fallando porque, para empezar, no sabía por qué funcionaba. Parecía funcionar, dadas las limitadas "pruebas" que Fred hacía, pero no era más que una coincidencia. Animado por una falsa seguridad, Fred cargó hacia el olvido. Ahora, puede que la gente más inteligente conozca a alguien como Fred, pero nosotros sabemos que las cosas no funcionan así. No dependemos de las coincidencias, ¿verdad?

A veces, puede que lo hagamos. En ocasiones, puede ser muy fácil confundir una feliz coincidencia con un plan deliberado. Vamos a ver algunos ejemplos.

Accidentes de implementación

Los accidentes de implementación son cosas que ocurren simplemente porque esa es la forma en que está escrito el código en ese momento. Acabamos dependiendo de condiciones de error o de contorno no documentadas.

Supongamos que llamamos a una rutina con datos malos. La rutina responde de una manera en particular y escribimos código en base a esa respuesta. Pero el autor no había planeado que esa rutina funcionase así; ni siquiera se había considerado. Cuando se "arregle" la rutina, puede que nuestro código se rompa. En el caso más extremo, puede que la rutina a la que hemos llamado ni siquiera esté diseñada para hacer lo que queremos, pero parece funcionar bien. Llamar a las cosas en el orden equivocado, o en el contexto equivocado, es un problema relacionado.

Aquí parece que Fred está intentando a la desesperada que salga algo en la pantalla utilizando un *framework* de renderización de GUI particular:

```
paint();
invalidate();
validate();
revalidate();
repaint();
paintImmediately();
```

Pero estas rutinas no se diseñaron para que se las llamase de esta manera; aunque parecen funcionar, en realidad es solo una coincidencia.

Para colmo de males, cuando la escena por fin se dibuja, Fred no va a volver atrás para eliminar las llamadas espurias. “Ahora funciona, mejor lo dejo como está...”.

Es fácil dejarse engañar por esta línea de pensamiento. ¿Por qué deberíamos correr el riesgo de enredar con algo que está funcionando? Bueno, se nos ocurren varias razones:

- Puede que en realidad no esté funcionado; tal vez solo lo parezca.
- La condición de contorno de la que dependemos puede ser solo un accidente. En circunstancias diferentes (una resolución de pantalla diferente, más núcleos de CPU), podría comportarse de manera distinta.
- Un comportamiento no documentado puede cambiar con el siguiente lanzamiento de biblioteca.
- Las llamadas adicionales e innecesarias ralentizan el código.
- Las llamadas adicionales incrementan el riesgo de introducir nuevos fallos propios.

Para código que escriba usted y que otros llamarán, todos los principios básicos de la buena modularidad y de la ocultación de la implementación tras interfaces pequeñas y bien documentadas pueden ayudar. Un contrato bien especificado (consulte el tema 23, “Diseño por contrato”) puede ayudar a eliminar los malentendidos.

Para las rutinas a las que llame usted, dependa solo de comportamientos documentados. Si no puede, por el motivo que sea, entonces documente bien su asunción.

Lo bastante cerca no existe

Una vez trabajamos en un proyecto grande que informaba acerca de datos introducidos desde una cantidad enorme de unidades de recopilación de datos de hardware sobre el terreno. La distribución de estas unidades abarcaba varios estados y zonas horarias y, por diversas razones logísticas e históricas, cada unidad estaba configurada en la hora local.[1] Como

resultado de las interpretaciones de las zonas horarias en conflictos y las inconsistencias en las políticas del horario de verano, los resultados estaban casi siempre mal, pero era solo un error por uno. Los desarrolladores del proyecto se habían acostumbrado a sumar o restar uno para obtener la respuesta correcta, con el razonamiento de que era solo un error por uno en esta situación. Y, después, la siguiente función vería que el valor estaba desviado por uno en el otro sentido y lo cambiaría de nuevo.

Pero el hecho de que fuese un error "solo" por uno parte del tiempo era una coincidencia, que enmascaraba un problema más profundo e importante. Sin un modelo apropiado de gestión de las horas, toda la base del código grande había empeorado con el tiempo hasta convertirse en una masa insostenible de sentencias `+1` y `-1`. Al final, no había nada correcto y el proyecto se abandonó.

Patrones fantasma

Los seres humanos estamos diseñados para ver patrones y causas, incluso cuando se trata solo de coincidencias. Por ejemplo, los líderes rusos siempre alternaban entre ser calvos y tener pelo: un líder del estado ruso calvo (o que estaba quedándose calvo) había sucedido a otro no calvo (con pelo), y viceversa, durante casi 200 años.[2]

Pero, aunque no escribiríamos código que dependiese de que el siguiente líder ruso sea calvo o tenga pelo, en algunos dominios pensamos de ese modo todo el tiempo. Los jugadores imaginan patrones en los números de la lotería, los juegos de dados o la ruleta cuando en realidad estos eventos son estadísticamente independientes. En las finanzas, el comercio de acciones y bonos está también plagado de coincidencias, más que de patrones reales y discernibles.

Un archivo de registro que muestra un error intermitente cada 1.000 solicitudes puede ser una condición de carrera difícil de diagnosticar o puede ser un fallo de los de toda la vida, sin más. Las pruebas que parecen pasarse en nuestro ordenador, pero no en el servidor, podrían indicar una diferencia entre los dos entornos, o quizá es solo una coincidencia. No lo dé por hecho, demuéstrelo.

Accidentes de contexto

También puede haber “accidentes de contexto”. Supongamos que estamos escribiendo un módulo de utilidad. Solo porque estemos escribiendo código para un entorno de GUI en este momento, ¿tiene que depender el módulo de que haya una GUI presente? ¿Dependemos de usuarios angloparlantes? ¿Usuarios alfabetizados? ¿De qué más dependemos que no esté garantizado?

¿Dependemos de que el directorio actual sea escribible? ¿De la presencia de determinadas variables de entorno o archivos de configuración? ¿De que la hora del servidor sea exacta, y dentro de qué tolerancia? ¿Dependemos de la velocidad y la disponibilidad de la red?

Cuando copiamos el código de la primera respuesta que encontramos en la red, ¿estábamos seguros de que el contexto era el mismo? ¿O estamos creando código tipo “culto cargo”, imitando solo la forma sin el contenido? [3]

Encontrar una respuesta que resulta que encaja no es lo mismo que encontrar la respuesta correcta.

Truco 62. No programe por casualidad.

Asunciones implícitas

Las coincidencias pueden ser engañosas en todos los niveles, desde la generación de requisitos hasta la realización de pruebas. Las pruebas están particularmente plagadas de causalidades falsas y resultados fortuitos. Es fácil asumir que X causa Y, pero, como decíamos en el tema 20 “Depuración”, no lo dé por hecho, demuéstrelo.

En todos los niveles, la gente opera con muchas asunciones en mente, pero estas rara vez están documentadas y, a menudo, están en conflicto entre diferentes desarrolladores. Las asunciones que no se basan en hechos bien establecidos son la pesadilla de todos los problemas.

Cómo programar de forma deliberada

Queremos pasar menos tiempo haciendo código como churros, encontrar y arreglar errores lo antes posible en el ciclo de desarrollo y crear menos errores, para empezar. Saber programar de forma deliberada puede ser de ayuda:

- Esté siempre atento a lo que está haciendo. Fred dejó que las cosas fuesen escapando de su control poco a poco hasta que acabó hervido como la rana del capítulo 1.
- ¿Puede explicar el código en detalle a un programador júnior? Si no es así, quizá esté dependiendo de casualidades.
- No escriba código a ciegas. Si construye una aplicación que no termina de comprender o utiliza una tecnología que no entiende, es probable que acabe topándose con casualidades. Si no está seguro de por qué funciona, no sabrá por qué falla.
- Trabaje a partir de un plan, da igual que ese plan esté en su cabeza, escrito en una servilleta o en una pizarra.
- Dependa solo de cosas fiables, no de asunciones. Si no está seguro de si algo es fiable, asuma lo peor.
- Documente sus asunciones. El tema 23, “Diseño por contrato”, puede ayudarle a aclarar sus asunciones en su mente, además de ayudarle a comunicárselas a otros.
- No pruebe solo su código, sino también sus asunciones. No haga suposiciones; pruébelas de verdad. Escriba una aserción para probar sus asunciones (consulte el tema 25, “Programación asertiva”). Si su aserción es correcta, habrá mejorado la documentación de su código. Si descubre que su asunción es errónea, considérese afortunado.
- Priorice su esfuerzo. Dedique tiempo a los aspectos importantes; es más que probable que sean las partes difíciles. Si no tienen cimientos o infraestructura correctos, todos los adornos serán irrelevantes.
- No sea esclavo de la historia. No permita que el código existente dicte el código futuro. Cualquier código puede sustituirse si ya no es apropiado. Incluso dentro de un programa, no permita que lo que ya ha hecho restrinja lo que hace a continuación; esté preparado para refactorizar (consulte el tema 40, “Refactorización”). Esta decisión puede afectar a la planificación del proyecto. La asunción es que el impacto será menor que el coste de no hacer el cambio.[4]

Así pues, la próxima vez que parezca que algo funciona, pero no sepa por qué, asegúrese de que no es solo una casualidad.

Las secciones relacionadas incluyen

- Tema 4, “Sopa de piedras y ranas hervidas”.
- Tema 9, “DRY: los males de la duplicación”.
- Tema 23, “Diseño por contrato”.
- Tema 34, “Estado compartido es estado incorrecto”.
- Tema 43, “Tenga cuidado ahí fuera”.

Ejercicios

Ejercicio 25

Un sistema de alimentación de datos de un proveedor le proporciona una matriz de tuplas que representan pares clave-valor. La clave de `DepositAccount` albergará una cadena del número de cuenta en el valor correspondiente:

```
[
...
{:DepositAccount, “564-904-143-00”}
...
]
```

Funcionaba a la perfección en las pruebas en los portátiles de 4 núcleos de los desarrolladores y en la máquina de construcción de 12 núcleos, pero, en los servidores de producción que se ejecutan en contenedores, todo el tiempo aparecen números de cuenta equivocados. ¿Qué está pasando?

Ejercicio 26

Está escribiendo código para un marcador automático para alertas de voz y tiene que gestionar una base de datos de información de contacto. La UIT especifica que los números de teléfono no deberían tener más de 15 dígitos, así que almacena el número de teléfono del contacto en un campo numérico que garantiza albergar al menos 15 dígitos. Lo ha probado de forma

exhaustiva por toda Norteamérica y todo parece ir bien, pero, de repente, recibe una serie de quejas de otras partes del mundo. ¿Por qué?

Ejercicio 27

Ha escrito una aplicación que adapta recetas comunes para el comedor de un crucero con capacidad para 5.000 personas, pero está recibiendo quejas de que las conversiones no son precisas. Lo comprueba y el código utiliza la fórmula de conversión de 16 tazas a un galón. Eso está bien, ¿no?

39 Velocidad de los algoritmos

En el tema 15, "Estimaciones", hemos hablado acerca de estimar cosas como cuánto se tarda en cruzar la ciudad a pie o cuánto llevará terminar un proyecto. Sin embargo, hay otro tipo de estimación que los programadores pragmáticos utilizan casi a diario: la estimación de los recursos que utilizan los algoritmos, como el tiempo, el procesador, la memoria, etc.

A menudo, este tipo de estimación es crucial. Si se le da a elegir entre dos maneras de hacer las cosas, ¿cuál escoge? Sabe durante cuánto tiempo se ejecuta su programa con 1.000 registros, ¿pero cómo escalará a 1.000.000? ¿Qué partes del código necesitan optimización?

Resulta que, con frecuencia, estas preguntas pueden responderse usando el sentido común, algo de análisis y una manera de escribir aproximaciones denominada notación Big O.

¿Qué queremos decir con estimar algoritmos?

La mayoría de los algoritmos no triviales manejan algún tipo de entrada de variables (ordenar `n` cadenas, invertir una matriz `m × n` o descifrar un mensaje con una clave de `n` bits). Por lo general, el tamaño de esta entrada afectará al algoritmo: cuanto mayor sea la entrada, mayor será el tiempo de ejecución o más memoria se utilizará.

Si la relación fuese siempre lineal (de manera que el tiempo aumentase en proporción directa al valor de `n`), esta sección no sería importante. Sin embargo, los algoritmos más relevantes no son lineales. La buena noticia es

que muchos son sublineales. Una búsqueda binaria, por ejemplo, no requiere mirar cada candidato cuando se encuentra una coincidencia. La mala noticia es que otros algoritmos son bastante peores que los lineales; los tiempos de ejecución o los requisitos de memoria aumentan mucho más rápido que n. Un algoritmo que tarda un minuto en procesar diez elementos puede tardar una vida en procesar 100.

Nos encontramos con que cada vez que escribimos cualquier cosa que contenga bucles o llamadas recursivas, comprobamos de forma subconsciente el tiempo de ejecución y los requisitos de memoria. Esto rara vez es un proceso formal, sino más bien una confirmación rápida de que lo que estamos haciendo es sensible a las circunstancias. Sin embargo, a veces nos descubrimos haciendo análisis más detallados. Ahí es donde la notación Big O resulta útil.

Notación Big O

La notación Big O, escrita $O()$, es una manera matemática de tratar con las aproxi- maciones. Cuando escribimos que una rutina de ordenamiento concreta ordena n registros en un tiempo $O(n^2)$, simplemente estamos diciendo que el tiempo en el peor de los casos variará como el cuadrado de n. Si se duplica el número de registros, el tiempo se multiplicará por cuatro aproximadamente. Piense que la O significa “del orden de”.

La notación $O()$ pone un límite superior en el valor de la cosa que estamos midiendo (tiempo, memoria, etc.) Si decimos que una función tarda un tiempo $O(n^2)$, sabemos que el límite superior del tiempo que tarda no va a ser más rápido que n^2. A veces, nos encontramos con funciones $O()$ bastante complejas, pero, como el término de orden más alto dominará el valor a medida que n aumente, la convención es eliminar todos los términos de orden inferior y no molestarse en mostrar ningún factor multiplicador constante:

$$O(n^2/2 + 3n) \text{ es igual que } O(n^2/2) \text{ es igual que } O(n^2).$$

En realidad, esta es una característica de la notación $O()$; un algoritmo $O(n^2)$ puede ser 1.000 veces más rápido que otro algoritmo $O(n^2)$, pero

no lo sabrá por la notación. Big O nunca va a proporcionarle cifras reales para el tiempo, la memoria o lo que sea: solo le dice cómo cambiarán estos valores a medida que cambie la entrada.

La figura 7.1 muestra varias notaciones $O()$ habituales con las que se encontrará, junto con un gráfico que compara tiempos de ejecución de algoritmos en cada categoría. Está claro que las cosas enseguida empiezan a irse de las manos una vez que se pasa de $O(n^2)$.

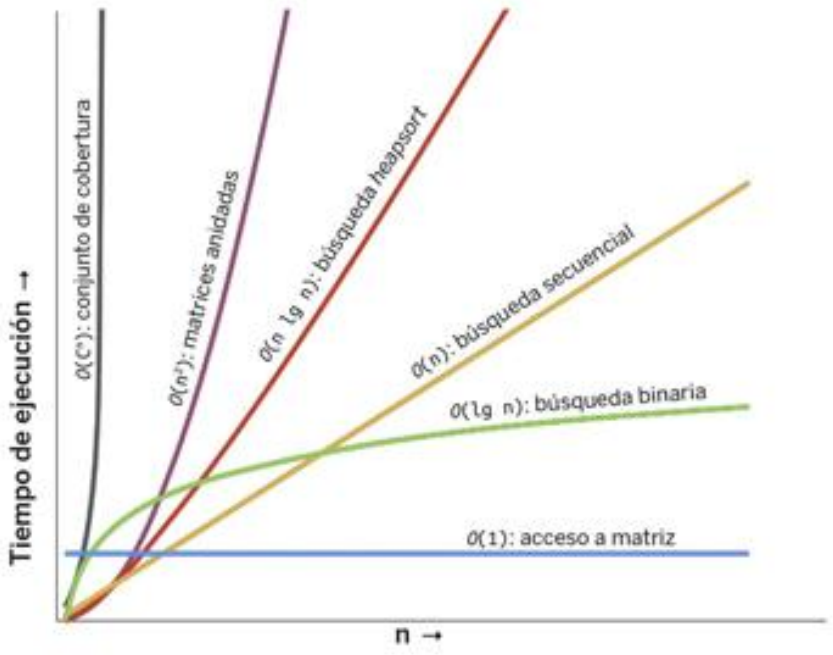


Figura 7.1. Tiempos de ejecución de varios algoritmos.

Por ejemplo, supongamos que tiene una rutina que tarda un segundo en procesar 100 registros. ¿Cuánto tardará en procesar 1.000? Si su código es $O(1)$, entonces seguirá tardando un segundo. Si es $O(\lg n)$, es probable que tenga que esperar unos tres segundos. $O(n)$ mostrará un incremento lineal a diez segundos, mientras que $O(n \lg n)$ tardará unos 33 segundos. Si tiene la mala suerte de tener una rutina $O(n^2)$, tendrá que esperar 100 segundos mientras hace sus cosas. Y, si está utilizando un algoritmo exponencial $O(2^n)$, puede que le convenga preparar café: la rutina debería terminar en unos 10263 años. Ya nos contará cómo acaba el universo.

La notación $O()$ no se aplica solo al tiempo; puede utilizarla para representar cualquier otro recurso utilizado por un algoritmo. Por ejemplo, a

menudo resulta útil para poder modelar el consumo de memoria (consulte los ejercicios para ver un ejemplo).

- $O(1)$: Constante (acceso a elemento de la matriz, sentencias simples).
- $O(\lg n)$: Logarítmico (búsqueda binaria). La base del logaritmo no importa, así que esto equivale a $O(\log n)$.
- $O(n)$: Lineal (búsqueda secuencial).
- $O(n \lg n)$: Peor que lineal, pero no mucho peor. (Tiempo de ejecución medio de ordenamiento rápido, *heapsort*).
- $O(n^2)$: Ley del cuadrado (ordenamientos por selección y por inserción).
- $O(n^3)$: Cúbico (multiplicación de dos matrices $n \times n$).
- $O(C^n)$: Exponencial (problema del viajante, partición de conjuntos).

Estimación con sentido común

Puede estimar el orden de muchos algoritmos básicos usando el sentido común.

- **Bucles simples:** Si un bucle simple se ejecuta de 1 a n, entonces es probable que el algoritmo sea $O(n)$; el tiempo se incrementa de forma lineal con n. Entre los ejemplos se incluyen las búsquedas, encontrar el valor máximo en una matriz o generar *checksums*.
- **Bucles anidados:** Si anida un bucle dentro de otro, entonces el algoritmo se convierte en $O(m \times n)$, donde m y n son los límites de los dos bucles. Esto ocurre con frecuencia en algoritmos de ordenamiento simple, como el ordenamiento de burbuja, donde el bucle exterior examina cada elemento de la matriz por turnos y el bucle interior determina dónde colocar ese elemento en el resultado ordenado. Los algoritmos de ordenamiento como este tienden a ser $O(n^2)$.
- **Búsqueda binaria:** Si su algoritmo reduce a la mitad el conjunto de cosas que considera cada vez que recorre el bucle, es probable que

sea logarítmico, $O(\lg n)$. Una búsqueda binaria de una lista ordenada, atravesar un árbol binario y encontrar el primer bit establecido en una palabra de la máquina pueden ser $O(\lg n)$.

- **Divide y vencerás:** Los algoritmos que dividen su trabajo de entrada en las dos mitades de forma independiente y luego combinan el resultado pueden ser $O(n \lg n)$. El ejemplo clásico es el ordenamiento rápido, que funciona dividiendo los datos en dos mitades y ordenando cada una de manera recursiva. Aunque técnicamente es $O(n^2)$, porque su comportamiento se degrada cuando se le introduce una entrada ordenada, el tiempo de ejecución medio del ordenamiento rápido es $O(n \lg n)$.
- **Combinatorio:** Cada vez que los algoritmos empiezan a fijarse en las permutaciones de las cosas, sus tiempos de ejecución pueden irse de las manos. Esto se debe a que las permutaciones implican factoriales (hay $5 ! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ permutaciones de los dígitos del 1 al 5). Cronometre un algoritmo combinatorio para cinco elementos: tardará seis veces más en ejecutarse para seis y 42 veces más para siete. Entre los ejemplos se incluyen algoritmos para muchos de los problemas difíciles reconocidos, como el problema del viajante, el empaquetamiento óptimo de cosas en un contenedor, la partición de un conjunto de números de manera que cada conjunto tenga el mismo total, etc. A menudo, se utiliza la heurística para reducir los tiempos de ejecución de estos tipos de algoritmos en dominios de problemas concretos.

Velocidad de los algoritmos en la práctica

No es probable que dedique mucho tiempo en su carrera a escribir rutinas de ordenamiento. Las que hay en las bibliotecas que tiene a su disposición superarán probablemente el rendimiento de cualquier cosa que pueda escribir sin un esfuerzo significativo. Sin embargo, los tipos de algoritmos básicos que acabamos de describir aparecen una y otra vez. Cada vez que se encuentra escribiendo un bucle simple, sabe que tiene un algoritmo $O(n)$. Si ese bucle contiene un bucle interior, entonces se trata de $O(m \times n)$. Debería estar preguntándose cómo de grandes pueden llegar

a ser esos valores. Si los números están limitados, entonces sabrá cuánto tardará el código en ejecutarse. Si los números dependen de factores externos (como el número de registros en una ejecución de lotes durante la noche o el número de nombres en una lista de personas), puede que le convenga detenerse y plantearse el efecto que esos valores grandes pueden tener en el tiempo de ejecución o el consumo de memoria.

Truco 63. Estime el orden de sus algoritmos.

Hay algunos enfoques que puede adoptar para abordar problemas potenciales. Si tiene un algoritmo que es $O(n^2)$, intente encontrar un enfoque divide y vencerás que le lleve a $O(n \lg n)$.

Si no está seguro de cuánto tardará su código o de cuánta memoria ocupará, intente ejecutarlo variando el número de registros de entrada o cualquier cosa que tenga probabilidades de afectar al tiempo de ejecución. Después, trace los resultados en un gráfico. Pronto debería hacerse una idea clara de la forma de la curva. ¿Se curva hacia arriba, es una línea recta o va aplanándose a medida que aumenta el tamaño de la entrada? Tres o cuatro puntos deberían darle una idea.

Plantéese también lo que está haciendo en el propio código. Un bucle $O(n^2)$ simple puede tener un rendimiento mejor que uno $O(n \lg n)$ complejo para valores más pequeños de n, sobre todo si el algoritmo $O(n \lg n)$ tiene un bucle interior caro.

En medio de toda esta teoría, no olvide que hay también consideraciones prácticas. Puede que el tiempo de ejecución parezca aumentar de forma lineal para conjuntos de entrada pequeños, pero, si introduce en el código millones de registros, de pronto el tiempo se degradará y el sistema empieza a fallar. Si prueba una rutina de ordenamiento con claves de entrada aleatorias, puede que se sorprenda la primera vez que encuentre una entrada ordenada. Intente cubrir tanto los aspectos teóricos como los prácticos. Después de todas estas estimaciones, el único tiempo que importa es la velocidad del código, ejecutándose en el entorno de producción, con datos reales. Esto nos lleva a nuestro siguiente truco.

Truco 64. Pruebe sus estimaciones.

Si es difícil conseguir tiempos precisos, utilice perfiladores de código para contar el número de veces que se ejecutan los diferentes pasos de su algoritmo y trace estas cifras en relación con el tamaño de la entrada.

El mejor no es siempre lo mejor

También necesita ser pragmático respecto a la elección de los algoritmos apropiados; el más rápido no siempre es el mejor para la tarea. Si se le da un conjunto de entrada pequeño, un ordenamiento por inserción directo tendrá un rendimiento tan bueno como el del ordenamiento rápido y le llevará menos tiempo escribirlo y depurarlo. Además, debe tener cuidado si el algoritmo que elige tiene un coste de configuración elevado. Para conjuntos de entrada pequeños, esta configuración puede empequeñecer el tiempo de ejecución y hacer que el algoritmo no sea apropiado.

Preste atención también a la optimización prematura. Siempre es buena idea asegurarse de que un algoritmo es de verdad un cuello de botella antes de dedicar un tiempo valioso a intentar mejorarlo.

Las secciones relacionadas incluyen

- Tema 15, “Estimaciones”.

Retos

- Todo desarrollador debería tener nociones acerca de cómo se diseñan y analizan los algoritmos. Robert Sedgewick ha escrito varios libros accesibles sobre la materia (*Algorithms* [SW11], *An Introduction to the Analysis of Algorithms* [SF13] y otros). Le recomendamos que añada uno de sus libros a su colección y ponga empeño en leerlo.
- Aquellos que quieran algo más detallado que lo que ofrece Sedgewick pueden leer los libros definitivos de Donald Knuth *El arte de programar ordenadores*, que analizan una amplia variedad de algoritmos.

- En el primer ejercicio que sigue nos fijamos en matrices de ordenamiento de enteros largos. ¿Cuál es el impacto si las claves son más complejas y la tara de la comparación de claves es alta? ¿Afecta la estructura de las claves a la eficiencia de los algoritmos de ordenamiento o el ordenamiento más rápido siempre es el más rápido?

Ejercicios

Ejercicio 28

Hemos creado código para un conjunto de rutinas de ordenamiento simples[5] en Rust. Ejecútelas en varios ordenadores a su disposición. ¿Siguen sus cifras las curvas esperadas? ¿Qué puede deducir acerca de las velocidades relativas de sus ordenadores? ¿Cuáles son los efectos de las distintas configuraciones para la optimización del compilador?

Ejercicio 29

En “Estimación con sentido común” decíamos que una búsqueda binaria es $O(\lg n)$. ¿Puede demostrarlo?

Ejercicio 30

En la figura 7.1 afirmábamos que $O(\lg n)$ es lo mismo que $O(\log 10 n)$ (o, en realidad, logaritmos de cualquier base). ¿Puede explicar por qué?

40 Refactorización

Cambio y decadencia en todo lo que veo...

—H. F. Lyte, Abide With Me

A medida que evoluciona un programa, va haciéndose necesario replantearse decisiones anteriores y reelaborar porciones del código. Este proceso es perfectamente natural. El código necesita evolucionar; no es una cosa estática.

Por desgracia, la metáfora más común para el desarrollo de software es la construcción de un edificio. El libro clásico de Bertrand Meyer, *Construcción de software orientado a objetos* [Mey97], utiliza el término "construcción de software" e incluso nosotros, los humildes autores, editamos la columna "*Software Construction*" para *IEEE Software* a principios de los años 2000.[6]

Pero usar la construcción como metáfora guía implica los siguientes pasos:

1. Un arquitecto dibuja los planos.
2. Los contratistas cavan para colocar los cimientos, construyen la superestructura, la instalación eléctrica y la fontanería y aplican los toques finales.
3. Los inquilinos se mudan allí y viven felices para siempre, llamando al servicio de mantenimiento del edificio para arreglar cualquier problema.

Bueno, el software no funciona exactamente así. Más que a la construcción, el software se parece a la jardinería; es más orgánico que el cemento. Plantamos muchas cosas en un jardín según un plan y unas condiciones iniciales. Algunas crecen muy bien y otras están destinadas a convertirse en abono. Puede que movamos algunas plantas con respecto a otras para aprovechar más la interacción de la luz y la sombra, el viento y la lluvia. Las plantas que crecen demasiado se dividen o se podan, y los colores que desentonan pueden trasladarse a ubicaciones más agradables a nivel estético. Arrancamos hierbajos y fertilizamos las plantas que necesitan algo de ayuda adicional. Vigilamos todo el tiempo la salud del jardín y hacemos ajustes (en el suelo, las plantas, la disposición) según sea necesario.

Las personas de negocios se sienten cómodas con la metáfora de la construcción de un edificio: es más científica que la jardinería, es repetible, hay una jerarquía rígida de creación de informes para la dirección, etc. Pero no estamos construyendo rascacielos; no estamos limitados por las restricciones de la física y el mundo real.

La metáfora de la jardinería se acerca mucho más a la realidad del desarrollo de software. Quizá una rutina específica se ha hecho demasiado

grande, o está intentando abarcar demasiado: es necesario dividirla en dos. Las cosas que no funcionan como se había planeado deben arrancarse o podarse.

Reescribir, reelaborar y rediseñar la arquitectura del código se conoce de forma colectiva como “reestructuración”, pero hay un subconjunto de esa actividad que se ha practicado como “refactorización”.

Martin Fowler define la refactorización en *Refactoring* [Fow19] como una:

> *técnica disciplinada para reestructurar un cuerpo de código existente, alterando su estructura interna sin cambiar su comportamiento externo.*

Las partes cruciales de esta definición son que:

1. La actividad es disciplinada, no la casa de Tócame Roque.
2. El comportamiento externo no cambia; no es el momento de añadir características.

La refactorización no está pensada para ser una actividad especial que se realiza muy de vez en cuando con mucha ceremonia, como levantar la tierra del jardín entero para replantar, sino que es una actividad del día a día, que avanza a pasos pequeños de bajo riesgo, más similar a arrancar hierbajos y pasar el rastrillo. En vez de una reescritura a gran escala y a lo loco de la base de código, se trata de un enfoque preciso y orientado para ayudar a que el código siga siendo fácil de cambiar.

Para garantizar que el comportamiento externo no ha cambiado, necesitamos buenas pruebas unitarias automatizadas que validen el comportamiento del código.

¿Cuándo deberíamos refactorizar?

Refactorizamos cuando hemos aprendido algo; cuando entendemos algo mejor de lo que lo hacíamos el año pasado, ayer o hace solo unos minutos.

Quizá se haya encontrado con un obstáculo porque el código ya no encaja del todo, o se da cuenta de que hay dos cosas que, en realidad, deberían estar fusionadas, o cualquier otra cosa que le dé la sensación de estar “mal”; no dude en cambiarlo. No deje para mañana lo que pueda hacer

hoy. Hay varias cosas que pueden hacer que un código sea un buen candidato a la refactorización:

- **Duplicación:** Ha descubierto una violación del principio DRY.
- **Diseño no ortogonal:** Ha descubierto algo que podría hacerse más ortogonal.
- **Información desfasada:** Las cosas cambian, los requisitos se desvían y su conocimiento del problema aumenta. El código necesita seguir el ritmo.
- **Uso:** A medida que personas reales en circunstancias reales van usando el sistema, se da cuenta de que ahora algunas características son más importantes de lo que había pensado en principio y características que parecían imprescindibles quizá no lo son.
- **Rendimiento:** Necesita pasar funcionalidad de un área del sistema a otra para mejorar el rendimiento.
- **Las pruebas se pasan:** Sí. En serio. Hemos dicho que la refactorización debería ser una actividad a pequeña escala, respaldada por pruebas buenas. Así pues, cuando ha añadido una pequeña cantidad de código y esa prueba extra se pasa, tiene una gran oportunidad de ponerse a limpiar lo que acaba de escribir.

Refactorizar su código (llevar la funcionalidad de un lado a otro y actualizar decisiones anteriores) es realmente un ejercicio de gestión del dolor. Afrontémoslo, cambiar el código fuente puede ser bastante doloroso: funcionaba, así que quizá sea mejor dejarlo quieto. Muchos desarrolladores se muestran reacios a reabrir una porción de código solo porque no esté del todo bien.

Complicaciones del mundo real

Vale, se acerca a sus compañeros de equipo o al cliente y dice: “Este código funciona, pero necesito otra semana para refactorizarlo por completo”.

No podemos reproducir aquí la respuesta que nos darían.

La presión del tiempo se utiliza a menudo como excusa para no refactorizar. Pero esta excusa no se sostiene: si no refactoriza ahora, tendrá

que invertir mucho más tiempo para arreglar el problema más adelante, cuando haya más dependencias con las que lidiar. ¿Tendrá entonces más tiempo a su disposición? No.

Puede que quiera explicar este principio a otros usando una analogía médica: piense en el código que necesita la refactorización como "un tumor". Extirparlo requiere cirugía invasiva. Puede hacerlo ahora y quitarlo mientras aún es pequeño o puede esperar mientras crece y se extiende, pero extirparlo entonces será más caro y más peligroso. Si espera todavía más, puede que pierda al paciente.

Truco 65. Refactorice pronto, refactorice con frecuencia.

Los daños colaterales en el código pueden ser igual de letales con el tiempo (consulte el tema 3, "Entropía del software"). La refactorización, como ocurre con la mayoría de las cosas, es más fácil de hacer cuando los problemas son pequeños, como una actividad regular mientras se escribe código. No debería necesitar "una semana para refactorizar" una porción de código; eso es reescribirla por completo. Si se necesita ese nivel de alteración, entonces puede que no sea capaz de hacerlo de inmediato. En vez de eso, asegúrese de que se incluye en la planificación. Asegúrese de que los usuarios del código afectado saben que la reescritura está en la agenda y cómo podría afectarles eso.

¿Cómo refactorizamos?

La refactorización surgió en la comunidad de Smalltalk y acababa de empezar a ganar popularidad cuando escribimos la primera edición de este libro, probablemente gracias al primer libro importante sobre el tema (*Refactoring: Improving the Design of Existing Code* [Fow19], que ahora está en su segunda edición).

En esencia, la refactorización es rediseño. Cualquier cosa que usted u otras personas de su equipo hayan diseñado puede rediseñarse teniendo en cuenta hechos nuevos, una comprensión más profunda, cambios en los requisitos, etc. Pero, si se pone a arrancar grandes cantidades de código sin

orden ni concierto, puede que acabe en una situación peor que en la que estaba cuando empezó.

Es evidente que la refactorización es una actividad que necesita abordarse de manera lenta, deliberada y cuidadosa. Martin Fowler ofrece los siguientes trucos acerca de cómo refactorizar sin causar más daños que beneficios:[7]

1. No intente refactorizar y añadir funcionalidad al mismo tiempo.
2. Asegúrese de que tiene buenas pruebas antes de empezar a refactorizar. Ejecute las pruebas con la mayor frecuencia posible. De ese modo, enseguida sabrá si los cambios han estropeado algo.
3. Avance a pasos pequeños y deliberados: mueva un campo de una clase a otra, divida un método, cambie el nombre a una variable. A menudo, la refactorización implica hacer muchos cambios localizados que tienen como resultado un cambio a mayor escala. Si avanza con pasos pequeños y hace pruebas después de cada paso, evitará una depuración prolongada.[8]

Refactorización automática

En la primera edición hablábamos de que “esta tecnología todavía tiene que aparecer fuera del mundo de Smalltalk, pero es probable que esto cambie...”. Y vaya si ha cambiado, porque ahora la refactorización automática está disponible en muchos IDE y para la mayoría de los lenguajes populares.
Estos IDE pueden cambiar el nombre a variables y métodos, dividir una rutina larga en varias más pequeñas, propagar de forma automática los cambios requeridos, arrastrar y soltar para ayudarnos a mover código, etc.

Hablaremos más sobre las pruebas a este nivel en el tema 41, “Probar para escribir código” y de las pruebas a mayor escala en la sección “Pruebas despiadadas y continuas”, en el tema 51, “Kit pragmático básico”, pero la idea del señor Fowler de mantener buenas pruebas de regresión es clave para una refactorización segura.

Si tiene que ir más allá de la refactorización y acaba cambiando el comportamiento externo o las interfaces, entonces puede que le ayude romper la construcción de forma deliberada: los clientes antiguos de este

código deberían fallar en la compilación. De ese modo, sabrá qué necesita actualizarse. La próxima vez que vea una porción de código que no sea exactamente como debería, arréglela. Gestione el dolor: si duele ahora, pero va a doler aún más en el futuro, es mejor que se ocupe de ello cuanto antes. Recuerde la lección del tema 3, “Entropía del software”: no viva con ventanas rotas.

Las secciones relacionadas incluyen

- Tema 3, “Entropía del software”.
- Tema 9, “DRY: los males de la duplicación”.
- Tema 12, “Balas trazadoras”.
- Tema 27, “No vaya más rápido que sus faros”.
- Tema 44, “Poner nombre a las cosas”.
- Tema 48, “La esencia de la agilidad”.

41 Probar para escribir código

La primera edición de este libro se escribió en tiempos más primitivos, cuando la mayoría de los desarrolladores no escribían pruebas; ¿para qué molestarse, si el mundo iba a acabarse en el año 2000, de todos modos?

En ese libro, teníamos una sección acerca de cómo crear código que fuese fácil de probar. Era una manera furtiva de convencer a los desarrolladores de que escribiesen pruebas.

En estos tiempos todos somos más sabios. Si todavía hay desarrolladores que no escriben pruebas, al menos saben que deberían hacerlo. Pero sigue habiendo un problema. Cuando preguntamos a los desarrolladores por qué escriben pruebas, nos miran como si acabásemos de preguntarles si aún escriben código usando tarjetas perforadas y dicen “para asegurarnos de que el código funciona”, con un “so tonto” al final que no llegan a decir. Y pensamos que eso es un error.

Entonces, ¿qué nos parece importante acerca de las pruebas? ¿Y cómo pensamos que habría que proceder al respecto?

Vamos a empezar por una afirmación atrevida:

Truco 66. Las pruebas no van de encontrar fallos.

Consideramos que los principales beneficios de las pruebas se dan cuando pensamos en ellas y las escribimos, no cuando las ejecutamos.

Pensar en las pruebas

Es lunes por la mañana y se acomoda para empezar a trabajar en un código nuevo. Tiene que escribir algo que consulte la base de datos para devolver una lista de personas que ven más de 10 vídeos a la semana en su sitio de "los vídeos de lavavajillas más divertidos del mundo".

```
def return_avid_viewers do
  # ... hmmm ...
end
```

¡Pare! ¿Cómo sabe que lo que está a punto de hacer es bueno?

La respuesta es que no puede saberlo. Nadie puede. Pero pensar en las pruebas puede hacer que sea más probable. Veamos cómo funciona.

Empiece por imaginar que ha acabado de escribir la función y ahora tiene que probarla. ¿Cómo lo haría? Bueno, le interesaría utilizar algunos datos de prueba, lo que probablemente signifique que le conviene trabajar en una base de datos que controle usted. Ahora algunos *frameworks* pueden gestionar eso por usted, ejecutando pruebas sobre una base de datos de prueba, pero en nuestro caso eso significa que deberíamos pasar la instancia de la base de datos a nuestra función en vez de usar una global, ya que eso nos permite cambiarla mientras se realizan las pruebas:

```
def return_avid_users(db) do
```

Después, habría que pensar en cómo poblar esos datos de prueba. El requisito pide una "lista de personas que ven más de 10 vídeos a la semana". Así pues, buscamos en el esquema de la base de datos campos que puedan ayudar. Encontramos dos campos con probabilidades en una tabla de "quién vio qué": `opened_video` y `completed_video`. Para escribir nuestros datos de prueba, necesitamos saber qué campo utilizar. Pero no sabemos qué significa el requisito y nuestro contacto del negocio

no está. Vamos a hacer trampas y a pasar el nombre del campo (lo que nos permitirá probar lo que tenemos y, potencialmente, cambiarlo más tarde):

```
def return_avid_users(db, qualifying_field_name) do
```

Hemos empezado por pensar en las pruebas y, sin escribir ni una línea de código, ya hemos hecho dos descubrimientos y los hemos utilizado para cambiar la API de nuestro método.

Escritura de código guiada por pruebas

En el ejemplo anterior, pensar en las pruebas ha hecho que reduzcamos el acoplamiento en nuestro código (al pasar una conexión a una base de datos en vez de utilizar una global) y aumentemos la flexibilidad (al convertir el nombre del campo que probamos en un parámetro). Pensar en escribir una prueba para nuestro método ha hecho que lo miremos desde fuera, como si fuésemos un cliente del código en vez de sus creadores.

Truco 67. Una prueba es la primera usuaria de su código.

Pensamos que es probable que este sea el mayor beneficio que ofrecen las pruebas: la realización de pruebas es un *feedback* crucial que guía la escritura del código.

Un método o función que tenga un acoplamiento fuerte con otro código es más difícil de probar, porque hay que configurar todo ese entorno antes de poder siquiera ejecutar el método. Por tanto, hacer que las cosas puedan probarse también reduce su acoplamiento.

Y antes de poder probar algo, tenemos que entenderlo. Parece una tontería, pero, en realidad, todos nos hemos lanzado de lleno a una porción de código basándonos en un entendimiento vago de lo que teníamos que hacer. Nos aseguramos a nosotros mismos que lo resolveremos sobre la marcha. Oh, y también añadiremos más tarde el código para soportar las condiciones de contorno. Oh, y el manejo de errores. Y el código acaba siendo cinco veces más grande de lo que debería porque está lleno de lógica condicional y casos especiales. Pero, si ilumina con la luz de una prueba ese código, todo se volverá más claro. Si piensa en las pruebas para las

condiciones de contorno y en cómo funcionarán antes de empezar a escribir el código, puede que encuentre los patrones en la lógica que simplificarán la función. Si piensa en las condiciones de error que necesitará probar, estructurará su función en consecuencia.

Desarrollo guiado por pruebas

Existe una escuela de programación que dice que, teniendo en cuenta todos los beneficios de pensar en las pruebas de antemano, ¿por qué no anticiparse y escribirlas también por adelantado? Practican el denominado "desarrollo guiado por pruebas" o TDD, por sus siglas en inglés. También lo encontrará con el nombre de "*test-first development*", "escribir las pruebas primero".[9]

El ciclo básico del TDD es:

1. Decidirse por una pequeña funcionalidad que queramos añadir.
2. Escribir una prueba que se pasará una vez que se implemente esa funcionalidad.
3. Ejecutar todas las pruebas. Verificar que el único fallo es que el que acabamos de escribir.
4. Escribir la cantidad de código más pequeña necesaria para conseguir que se pase la prueba y verificar que ahora todas las pruebas se ejecutan con limpieza.
5. Refactorizar nuestro código: comprobar si hay una manera de mejorar lo que acabamos de escribir (la prueba o la función). Asegurarnos de que las pruebas siguen pasándose cuando hemos acabado.

La idea es que el ciclo debería ser muy corto: cuestión de minutos, de manera que estemos constantemente escribiendo pruebas y, después, haciendo que funcionen.

Consideramos que el TDD tiene un beneficio importante para la gente que está iniciándose en la creación de pruebas. Si sigue el flujo de trabajo del TDD, garantizará que siempre tiene pruebas para su código, y eso significa que siempre estará pensando en sus pruebas. Sin embargo, también

hemos visto a personas que se han convertido en esclavas del TDD. Esto se manifiesta de diversas formas:

- Dedican una cantidad de tiempo excesiva a asegurarse de que siempre tienen una cobertura de pruebas del 100 %.
- Tienen muchas pruebas redundantes. Por ejemplo, antes de escribir una clase por primera vez, los seguidores del TDD escribirán primero una prueba que falle que solo hace referencia al nombre de la clase. Falla y, entonces, escriben una definición de clase y se pasa. Pero ahora tienen una prueba que no hace absolutamente nada; la siguiente prueba que escriben también hará referencia a la clase, lo que hace que la primera sea innecesaria. Hay más cosas que modificar si el nombre de clase se cambia más adelante. Y este es solo un ejemplo trivial.
- Sus diseños tienden a empezar desde abajo e ir hacia arriba. (Consulte el cuadro "*Bottom-up vs. top-down vs.* cómo debería hacerlo" que veremos enseguida).

Por supuesto, practique el TDD, pero, si lo hace, no olvide parar de vez en cuando y analizar el panorama general. Es fácil verse seducido por el mensaje `"tests passed"` en vez de escribiendo un montón de código que, en realidad, no nos acerca más a una solución.

TDD: necesita saber adónde va

Un viejo chiste pregunta: "¿Cómo te comes un elefante?". El remate es: "Bocado a bocado". Y esta idea se vende a menudo como un beneficio del TDD. Cuando no pueda comprender un problema en su conjunto, avance a pasos pequeños, prueba a prueba. Sin embargo, este enfoque puede resultar engañoso al animarle a centrarse en los problemas fáciles y pulirlos hasta el infinito mientras ignora la razón real por la que está escribiendo el código. Un ejemplo interesante de esto sucedió en 2006, cuando Ron Jeffries, una figura destacada del movimiento ágil, comenzó una serie de publicaciones en un blog que documentaban su creación guiada por pruebas del código de un solucionador de *sudoku*.[10] Después de cinco publicaciones, había perfeccionado la representación del tablero subyacente, refactorizando

varias veces hasta que estuvo satisfecho con el modelo de objeto. Pero después abandonó el proyecto. Es interesante leer las publicaciones en el blog y ver cómo una persona inteligente se desvía de la cuestión principal por culpa de pequeñeces, deslumbrada por el brillo de las pruebas que se pasan.

Bottom-up vs. top-down vs. cómo debería hacerlo

Cuando la informática era joven y despreocupada, había dos escuelas de diseño: *top-down* (descendente) y *bottom-up* (ascendente). Los seguidores del modelo *top-down* decían que deberíamos empezar por el problema general que estamos intentando resolver y descomponerlo en un número pequeño de partes. Después, esas partes se descompondrían en porciones más pequeñas, y así sucesivamente, hasta que acabásemos con partes lo bastante pequeñas para expresarlas en código. Los partidarios del modelo *bottom-up* construyen código como se construiría una casa. Empiezan por la parte inferior, creando una capa de código que les proporciona ciertas abstracciones que están más cerca del problema que están intentando resolver. Después, añaden otra capa, con abstracciones de un nivel más alto. Siguen haciéndolo hasta que la capa final es una abstracción que soluciona el problema. "Hágalo así...".

Ninguna de las escuelas funciona en realidad, porque ambas ignoran uno de los aspectos más importantes del diseño de software: no sabemos lo que estamos haciendo cuando empezamos. Los seguidores del modelo *top-down* asumen que pueden expresar el requisito completo de antemano: no pueden. Los adeptos del *bottom-up* asumen que pueden construir una lista de abstracciones que, al final, les llevará a una única solución de alto nivel, pero ¿cómo pueden decidir acerca de la funcionalidad de las capas cuando no saben hacia dónde se dirigen?

Truco 68. Construya de extremo a extremo, no de arriba abajo o de abajo arriba.

Creemos firmemente que la única manera de construir software es hacerlo de forma gradual. Cree porciones pequeñas de funcionalidad de extremo a extremo, aprendiendo acerca del problema a medida que avanza. Aplique este aprendizaje mientras va completando el código, implique al cliente en cada paso y haga que guíe el proceso.

Como contraste, Peter Norvig describe un enfoque alternativo[11] que parece tener un carácter muy diferente: en vez de dejar que le guíen las pruebas, comienza con un entendimiento básico de cómo se resuelve

tradicionalmente este tipo de problemas (usando la propagación de restricciones), y después se concentra en perfeccionar su algoritmo. Aborda la representación del tablero en una docena de líneas de código que fluyen directamente desde su explicación sobre la notación.

Está claro que las pruebas pueden ayudar a guiar el desarrollo, pero, como ocurre con cualquier guía, a menos que tengamos un destino en mente, podemos acabar yendo en círculos.

De vuelta al código

El desarrollo basado en componentes es desde hace mucho tiempo un objetivo noble del desarrollo de software.[12] La idea es que los componentes de software genéricos deberían estar disponibles y combinarse con la misma facilidad con la que se combinan los circuitos integrados (CI) comunes. Pero esto solo funciona si se sabe que los componentes que se utilizan son fiables y si tenemos voltajes comunes, estándares de interconexión, temporización, etc.

Los chips están diseñados para probarse no solo en la fábrica, no solo cuando se instalan, sino también sobre el terreno cuando se despliegan. Los chips y sistemas más complejos pueden tener una función completa de autocomprobación incorporada (*Built-In Self Test*, BIST) que ejecuta de manera interna algunos diagnósticos de nivel básico o un mecanismo de acceso a pruebas (*Test Access Mechanism*, TAM) que ofrece un arnés de pruebas que permite al entorno externo proporcionar estímulos y recoger respuestas del chip. Podemos hacer lo mismo con el software. Al igual que nuestros compañeros del hardware, necesitamos integrar en el software la capacidad para probarse desde el principio y probar cada pieza de manera exhaustiva antes de intentar unirla a las demás.

Pruebas unitarias

Las pruebas a nivel de chip para el hardware son más o menos equivalentes a las pruebas unitarias en el software: pruebas que se hacen en cada módulo, en aislamiento, para verificar su comportamiento. Podemos hacernos una idea más clara de cómo reaccionará un módulo en el amplio

mundo una vez que lo hayamos probado en condiciones controladas (incluso artificiosas). Una prueba unitaria de software es código que ejercita un módulo. Por lo general, la prueba unitaria establecerá algún tipo de entorno artificial y, después, invocará rutinas en el módulo que está probándose. A continuación, podrá comprobar los resultados que se devuelven, ya sea comparándolos con valores conocidos o con los resultados de ejecuciones anteriores de la misma prueba (pruebas de regresión). Después, cuando montemos nuestro "Software-IC" para formar un sistema completo, tendremos la tranquilidad de que las piezas individuales van a funcionar como se espera y, luego, podremos usar las mismas instalaciones de pruebas unitarias para probar el sistema como conjunto. Hablaremos de esta comprobación del sistema a gran escala en "Pruebas despiadadas y continuas", en el tema 51, "Kit pragmático básico".

Antes de llegar tan lejos, sin embargo, necesitamos decidir qué probar a nivel unitario. Históricamente, los programadores echaban un puñado de bits de datos aleatorios al código, miraban las sentencias impresas y lo consideraban probado. Podemos hacerlo mucho mejor.

Pruebas en relación con el contrato

Nos gusta imaginar las pruebas unitarias como pruebas en relación con el contrato (consulte el tema 23, "Diseño por contrato"). Queremos escribir casos de prueba que garanticen que una unidad dada honra su contrato. Eso nos dirá dos cosas: si el código cumple el contrato y si el contrato significa lo que nosotros creemos. Queremos probar si el módulo proporciona la funcionalidad que promete, sobre una amplia variedad de casos de prueba y condiciones de contorno.

¿Qué significa esto en la práctica? Vamos a empezar por un ejemplo numérico sencillo: una rutina de raíz cuadrada. Su contrato documentado es simple:

```
precondiciones:
   argument >= 0;

postcondiciones:
   ((result * result)–argument).abs <= epsilon*argument;
```

Esto nos dice qué probar:

- Pasar un argumento negativo y asegurarse de que se rechaza.
- Pasar un argumento de cero y asegurarse de que se acepta (este es el valor de contorno).
- Pasar valores entre cero y el argumento máximo expresable y verificar que la diferencia entre el cuadrado del resultado y el argumento original es menor que alguna fracción pequeña del argumento (épsilon).

Armados con este contrato y asumiendo que nuestra rutina hace su propia comprobación de las precondiciones y las postcondiciones, podemos escribir un *script* de pruebas básico para ejercitar la función de la raíz cuadrada.

Después, podemos llamar a esta rutina para probar nuestra función de raíz cuadrada:

```
assertWithinEpsilon(my_sqrt(0), 0)
assertWithinEpsilon(my_sqrt(2.0), 1.4142135624)
assertWithinEpsilon(my_sqrt(64.0), 8.0)
assertWithinEpsilon(my_sqrt(1.0e7), 3162.2776602)
assertRaisesException fn => my_sqrt(-4.0) end
```

Se trata de una prueba bastante simple; en el mundo real, cualquier módulo no trivial tiene probabilidades de depender de otros módulos, así que ¿cómo abordamos la tarea de probar la combinación?

Supongamos que tenemos un módulo `A` que utiliza `DataFeed` y `LinearRegression`. En orden, probaríamos:

1. El contrato de `DataFeed` de principio a fin.
2. El contrato de `LinearRegression` de principio a fin.
3. El contrato de `A`, que depende de los otros contratos, pero no los expone de manera directa.

Este estilo de prueba requiere que probemos primero los subcomponentes de un módulo. Una vez que se han verificado esos componentes, puede probarse el módulo en sí.

Si las pruebas de `DataFeed` y `LinearRegression` se pasan, pero la prueba de `A` falla, podemos estar bastante seguros de que el problema está

en A, o en el uso que hace A de uno de esos subcomponentes. Esta técnica es una manera genial de reducir el trabajo de depuración: enseguida podemos concentrarnos en la fuente probable del problema dentro del módulo A, y no perder el tiempo reexaminando sus componentes.

¿Por qué nos tomamos tantas molestias? Sobre todo, queremos evitar crear una "bomba de relojería", algo que pasa desapercibido y explota en el momento más inoportuno más adelante en el proyecto. Al enfatizar las pruebas en relación con el contrato, podemos intentar evitar tantos de esos desastres en etapas posteriores como sea posible.

Truco 69. Diseñe para probar.

Pruebas *ad hoc*

Las pruebas *ad hoc* son aquellas en las que hurgamos en nuestro código de forma manual. Pueden ser algo tan sencillo como un `console.log()`, o una porción de código introducida de manera interactiva en un depurador, entorno IDE o un bucle REPL.

Al final de la sesión de depuración, necesita formalizar esta prueba *ad hoc*. Si el código se ha estropeado una vez, es probable que vuelva a hacerlo. No deseche sin más la prueba que ha creado; añádala a su arsenal de pruebas unitarias existentes.

Construya una ventana de pruebas

Incluso los mejores conjuntos de pruebas tienen pocas probabilidades de encontrar todos los fallos; hay algo en las condiciones húmedas y cálidas del entorno de producción que parece hacer que salgan a la luz. Eso significa que, a menudo, necesitaremos probar una porción del código una vez que se ha desplegado, con datos del mundo real corriendo por sus venas. A diferencia de una placa de circuito impreso o un chip, no tenemos pines de prueba en el software, pero podemos proporcionar varias vistas del estado interno de un módulo, sin usar el depurador (algo que puede ser inconveniente o imposible en una aplicación de producción).

Los archivos de registro que contienen mensajes de seguimiento son uno de esos mecanismos. Los mensajes de registro deberían tener un formato regular y coherente; puede que desee analizarlos sintácticamente de manera automática para deducir el tiempo de procesamiento o las rutas lógicas que ha seguido el programa. Los diagnósticos con formatos malos o incoherentes no son más que "palabrería": son difíciles de leer y analizarlos sintácticamente es poco práctico.

Otro mecanismo para entrar en un código en ejecución es la secuencia del "atajo de teclado" o URL mágica. Cuando se pulsa esta combinación de teclas concreta o se accede a la URL, aparece una ventana de control de diagnóstico con mensajes de estado, etc. Por lo general, esto no es algo que mostraríamos a los usuarios finales, pero puede resultar muy útil para el soporte técnico.

A nivel más general, podríamos utilizar un *feature switch* para habilitar diagnósticos adicionales para un usuario o una clase de usuarios en particular.

Una cultura de pruebas

Todo el software que escribimos será probado (si no por nosotros y nuestro equipo, por los usuarios finales), así que es mejor planificar las pruebas de forma exhaustiva. Algo de planificación puede ser una gran ayuda para minimizar los costes de mantenimiento y las llamadas al servicio técnico.

En realidad, tenemos pocas opciones:

- Probar primero.
- Probar durante.
- No probar nunca.

"Probar primero", incluyendo el diseño guiado por pruebas, es probablemente la mejor opción en la mayoría de las circunstancias, ya que garantiza que las pruebas se realizan. Pero, a veces, esto no resulta conveniente o útil, así que "Probar durante" puede ser un buen plan B, donde escribimos algo de código, jugueteamos con él, escribimos sus pruebas y, después, pasamos al siguiente fragmento. La peor opción se

denomina “Probar más tarde”, pero ¿a quién queremos engañar? “Probar más tarde” en realidad significa “No probar nunca”.

Una cultura de pruebas significa que todas las pruebas se pasan todo el tiempo. Ignorar una avalancha de pruebas que “siempre fallan” hace que sea más fácil ignorar todas las pruebas, y comienza el círculo vicioso (consulte el tema 3, “Entropía del software”).

Una confesión

Yo (Dave) soy conocido por decirle a la gente que ya no escribo pruebas. En parte, lo hago para sacudir la fe de aquellos que han convertido las pruebas en una religión y, en parte, lo digo porque es verdad (en cierto modo).

Llevo 45 años escribiendo código, y más de 30 de ellos escribiendo pruebas automatizadas. Pensar en las pruebas forma parte integral de la manera en que enfoco la creación del código. Me sentía cómodo. Y mi personalidad insiste en que, cuando algo empieza a resultarme cómodo, debería pasar a otra cosa.

En este caso, decidí dejar de escribir pruebas durante un par de meses y ver cómo afectaba eso a mi código. Para mi sorpresa, la respuesta fue “no mucho”, así que dediqué algo de tiempo a averiguar por qué.

Creo que la respuesta es que (para mí) la mayoría de los beneficios de las pruebas vienen de pensar en las pruebas y su impacto en el código y, después de hacerlo durante tanto tiempo, podría hacer la parte de pensar sin escribir en realidad las pruebas. Mi código todavía tenía la capacidad para probarse, pero no se probaba, sin más. Pero eso ignora el hecho de que las pruebas también son una manera de comunicarse con otros desarrolladores, así que ahora escribo pruebas para código compartido con otros o que depende de las peculiaridades de dependencias externas.

Andy dice que no debería incluir esta nota. Le preocupa que podría tentar a los desarrolladores con poca experiencia a no hacer pruebas. Esto es lo que propongo: ¿Debería usted escribir pruebas? Sí. Pero, cuando lleve 30 años haciéndolo, atrévase a experimentar un poco para ver dónde está el beneficio para usted.

Trate el código de pruebas con el mismo cuidado que cualquier código de producción. Manténgalo desacoplado, limpio y robusto. No dependa de cosas poco fiables (consulte el tema 38, “Programar por casualidad”) como la posición absoluta de los *widgets* en un sistema de GUI, los sellos de tiempo exactos de un registro del servidor o las palabras exactas de los mensajes de error. Crear pruebas para este tipo de cosas tendrá como resultado pruebas frágiles.

Truco 70. Pruebe su software, o lo harán sus usuarios.

No se equivoque, la creación de pruebas es parte de la programación. No es algo que haya que dejar en manos de otros departamentos o personas.

Las pruebas, el diseño, la escritura del código... Todo es programación.

Las secciones relacionadas incluyen

- Tema 27, “No vaya más rápido que sus faros”.
- Tema 51, “Kit pragmático básico”

42 Pruebas basadas en propiedades

Доверяй но проверяй (Confía, pero verifica).

—Proverbio ruso

Recomendamos escribir pruebas unitarias para sus funciones. Puede hacerlo pensando en las cosas típicas que podrían ser un problema, basándose en sus conocimientos acerca de aquello que está probando.

Sin embargo, hay un problema pequeño, pero que puede ser significativo, acechando en ese párrafo. Si usted escribe el código original y escribe las pruebas, ¿es posible que una asunción incorrecta pueda expresarse en ambos? El código pasa las pruebas porque hace lo que se supone que tiene que hacer según su comprensión.

Una manera de evitar esto es hacer que personas diferentes escriban las pruebas y el código que se va a probar, pero no nos gusta esto: como hemos dicho en el tema 41, “Probar para escribir código”, uno de los mayores beneficios de pensar en las pruebas es la manera en que eso aporta información al código que escribimos. Perdemos eso cuando el trabajo de la creación de pruebas se separa del de la escritura del código.

En vez de eso, preferimos una alternativa en la que el ordenador, que no comparte sus ideas preconcebidas, hace parte de las pruebas por usted.

Contratos, invariantes y propiedades

En el tema 23, “Diseño por contrato”, hablábamos de la idea de que el código tiene contratos que cumple: cumplimos las condiciones cuando introducimos la entrada y nos dará ciertas garantías acerca de los resultados que produce.

También hay invariantes de código, cosas que siguen siendo ciertas acerca de alguna parte del estado cuando se pasa a través de una función. Por ejemplo, si ordenamos una lista, el resultado tendrá el mismo número de elementos que la original; la longitud es invariante.

Una vez que determinamos nuestros contratos e invariantes (que vamos a agrupar y a llamar “propiedades”) podemos utilizarlos para automatizar nuestras pruebas. Así, tenemos lo que se denomina “pruebas basadas en propiedades”.

Truco 71. Use pruebas basadas en propiedades para validar sus asunciones.

Como ejemplo artificial, podemos crear algunas pruebas para nuestra lista ordenada. Ya hemos establecido una propiedad: la lista ordenada tiene el mismo tamaño que la original. También podemos afirmar que ningún elemento del resultado puede ser mayor que el que le sigue.

Ahora podemos expresar eso en código. La mayoría de los lenguajes tienen algún tipo de *framework* para pruebas basadas en propiedades. Este ejemplo está en Python y utiliza la herramienta Hypothesis y pytest, pero los principios son bastante universales.

Esta es la fuente completa de las pruebas:

proptest/sort.py

```
from hypothesis import given
import hypothesis.strategies as some

@given(some.lists(some.integers()))
def test_list_size_is_invariant_across_sorting(a_list):
  original_length = len(a_list)
  a_list.sort()
  assert len(a_list) == original_length

@given(some.lists(some.text()))
def test_sorted_result_is_ordered(a_list):
```

```
  a_list.sort()
  for i in range(len(a_list)–1):
    assert a_list[i] <= a_list[i + 1]
```

Esto es lo que pasa cuando la ejecutamos:

```
$ pytest sort.py
======================          test          session          starts
=======================
...
plugins: hypothesis-4.14.0

sort.py .. [100%]
====================      2      passed      in      0.95      seconds
====================
```

Aquí no hay mucho drama, pero, entre bastidores, Hypothesis ha ejecutado nuestras dos pruebas cien veces, pasando una lista diferente cada vez. Las listas tendrán longitudes variadas y contenidos diferentes. Es como si hubiésemos creado 200 pruebas individuales con 200 listas aleatorias.

Generación de datos de prueba

Como la mayoría de bibliotecas de pruebas basadas en propiedades, Hypothesis nos proporciona un minilenguaje para describir los datos que debería generar. El lenguaje se basa en llamadas a funciones en el módulo `hypothesis.strategies`, al que hemos puesto el alias `some`, porque se lee mejor. Si escribiésemos:

```
@given(some.integers())
```

Nuestra función de prueba se ejecutaría múltiples veces. Cada vez se le pasaría un entero diferente. Si, en vez de eso, escribiésemos lo siguiente:

```
@given(some.integers(min_value=5,  max_value=10).map(lambda  x:  x  *
2))
```

recibiríamos los números pares entre 10 y 20.

También podemos componer tipos, de manera que:

```
@given(some.lists(some.integers(min_value=1), max_size=100))
```

serán listas de números naturales que tienen una longitud de 100 elementos como máximo.

Esto no es un tutorial acerca de ningún *framework* concreto, así que vamos a saltarnos varios detalles chulos y a fijarnos en un ejemplo del mundo real.

Encontrar asunciones malas

Estamos escribiendo un sistema simple de procesamiento de pedidos y control del *stock* (porque siempre hay sitio para uno más). Modela los niveles de *stock* con un objeto `Warehouse`. Podemos consultar a un almacén para ver si hay algo en *stock*, retirar cosas del *stock* y recibir los niveles de *stock* actuales.

Este es el código:

proptest/stock.py

```
class Warehouse:
  def __init__(self, stock):
    self.stock = stock

  def in_stock(self, item_name):
    return (item_name in self.stock) and (self.stock[item_name] >
0)

  def take_from_stock(self, item_name, quantity):
    if quantity <= self.stock[item_name]:
      self.stock[item_name] -= quantity
    else:
      raise Exception("Oversold {}".format(item_name))

  def stock_count(self, item_name):
      return self.stock[item_name]
```

Hemos escrito una prueba unitaria básica, que se pasa:

proptest/stock.py

```
def test_warehouse():
  wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
  assert wh.in_stock("shoes")
```

```
  assert wh.in_stock(“hats”)
  assert not wh.in_stock(“umbrellas”)

  wh.take_from_stock(“shoes”, 2)
  assert wh.in_stock(“shoes”)

  wh.take_from_stock(“hats”, 2)
  assert not wh.in_stock(“hats”)
```

Después, hemos escrito una función que procesa una solicitud para pedir artículos del almacén. Devuelve una tupla donde el primer elemento es, bien “ok”, bien “not available”, seguido del artículo y la cantidad solicitada. También hemos escrito algunas pruebas y se pasan:

proptest/stock.py

```
def order(warehouse, item, quantity):
  if warehouse.in_stock(item):
    warehouse.take_from_stock(item, quantity)
    return ( “ok”, item, quantity )
  else:
    return ( “not available”, item, quantity )
```

proptest/stock.py

```
def test_order_in_stock():
  wh = Warehouse({“shoes”: 10, “hats”: 2, “umbrellas”: 0})
  status, item, quantity = order(wh, “hats”, 1)
  assert status == “ok”
  assert item == “hats”
  assert quantity == 1
  assert wh.stock_count(“hats”) == 1

def test_order_not_in_stock():
  wh = Warehouse({“shoes”: 10, “hats”: 2, “umbrellas”: 0})
  status, item, quantity = order(wh, “umbrellas”, 1)
  assert status == “not available”
  assert item == “umbrellas”
  assert quantity == 1
  assert wh.stock_count(“umbrellas”) == 0

def test_order_unknown_item():
  wh = Warehouse({“shoes”: 10, “hats”: 2, “umbrellas”: 0})
  status, item, quantity = order(wh, “bagel”, 1)
  assert status == “not available”
```

```
   assert item == “bagel”
   assert quantity == 1
```

En la superficie, todo parece estar bien, pero, antes de enviar el código, vamos a añadir algunas pruebas basadas en propiedades.

Una cosa que sabemos es que el *stock* no puede aparecer y desaparecer a través de nuestra transacción. Eso significa que, si tomamos algunos artículos del almacén, la cantidad que tomamos más la cantidad que hay en ese momento en el almacén debería ser igual a la cantidad que había originalmente en el almacén. En la siguiente prueba, ejecutamos nuestra prueba con el parámetro del artículo elegido de manera aleatoria entre `“hat”` (sobrero) o `“shoe”` (zapato) y la cantidad elegida entre 1 y 4:

proptest/stock.py

```
@given(item = some.sampled_from([“shoes”, “hats”]),
     quantity = some.integers(min_value=1, max_value=4))

def
test_stock_level_plus_quantity_equals_original_stock_level(item,
quantity):
   wh = Warehouse({“shoes”: 10, “hats”: 2, “umbrellas”: 0})
   initial_stock_level = wh.stock_count(item)
   (status, item, quantity) = order(wh, item, quantity)
   if status == “ok”:
      assert wh.stock_count(item) + quantity == initial_stock_level
```

Vamos a ejecutarla:

```
$ pytest stock.py
. . .
stock.py:72:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
stock.py:76:                                                          in
test_stock_level_plus_quantity_equals_original_stock_level
   (status, item, quantity) = order(wh, item, quantity)
stock.py:40: in order
   warehouse.take_from_stock(item, quantity)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <stock.Warehouse object at 0x10cf97cf8>, item_name = ‘hats’
quantity = 3

   def take_from_stock(self, item_name, quantity):
```

```
    if quantity <= self.stock[item_name]:
    self.stock[item_name] -= quantity
  else:
    raise Exception("Oversold {}".format(item_name))
    Exception: Oversold hats

stock.py:16: Exception
———————————— Hypothesis ————————————
Falsifying example:
test_stock_level_plus_quantity_equals_original_stock_level(
item='hats', quantity=3)
```

Salta por los aires en `warehouse.take_from_stock`: hemos intentado retirar tres sombreros del almacén, pero solo tiene dos en *stock*.

Nuestra prueba de propiedades ha encontrado una asunción incorrecta: nuestra función `in_stock` solo comprueba que hay al menos uno de los artículos dados en *stock*, pero lo que necesitamos es asegurarnos de que tenemos suficientes para completar el pedido:

proptest/stock1.py

```
def in_stock(self, item_name, quantity):
▶                return     (item_name     in     self.stock)     and
(self.stock[item_name] >= quantity)
```

Y cambiamos también la función `order`:

proptest/stock1.py

```
def order(warehouse, item, quantity):
▶         if warehouse.in_stock(item, quantity):
    warehouse.take_from_stock(item, quantity)
      return ( "ok", item, quantity )
  else:
    return ( "not available", item, quantity )
```

Y ahora nuestra prueba de propiedades se pasa.

Las pruebas basadas en propiedades nos sorprenden a menudo

En el ejemplo anterior, hemos utilizado una prueba basada en propiedades para comprobar que los niveles de *stock* se ajustaban de manera adecuada. La prueba ha encontrado un fallo, pero no tenía nada que ver con el ajuste de los niveles de *stock*, sino que era un fallo en nuestra función `in_stock`.

Este es el poder y la frustración de las pruebas basadas en propiedades. Son poderosas porque establecemos algunas reglas para generar entradas, establecemos aserciones para validar la salida y, después, las dejamos trabajar a su aire. Nunca sabemos con seguridad lo que va a pasar. Puede que la prueba se pase, que una aserción falle o que el código falle por completo porque no puede gestionar las entradas que se le han dado.

La frustración surge porque puede ser difícil determinar qué ha fallado.

Nuestra sugerencia es que, cuando una prueba basada en propiedades falle, averigüe qué parámetros estaba pasando a la función de prueba y, después, utilice esos valores para crear una prueba unitaria corriente aparte. Esa prueba unitaria hará dos cosas por usted. Primero, le permitirá centrarse en el problema sin todas las llamadas adicionales que hace en su código el *framework* de pruebas basadas en propiedades. En segundo lugar, esa prueba unitaria actúa como una prueba de regresión. Como las pruebas basadas en propiedades generan valores aleatorios que se pasan a su prueba, no hay garantías de que se utilicen los mismos valores la próxima vez que ejecute pruebas. Tener una prueba unitaria que obligue a que se utilicen esos valores garantiza que ese fallo no se colará.

Las pruebas basadas en propiedades también ayudan al diseño

Cuando hablábamos de las pruebas unitarias, decíamos que uno de sus principales beneficios era la manera en que nos hacían pensar en el código: una prueba unitaria es la primera clienta de nuestra API.

Lo mismo se aplica a las pruebas basadas en propiedades, pero de una forma un poco diferente. Nos hacen pensar en nuestro código en relación a las invariantes y los contratos; pensamos en lo que no debe cambiarse y en lo que debe ser cierto. Este entendimiento tiene un efecto mágico en nuestro

código, eliminando casos límite y resaltando funciones que dejan los datos en un estado inconsistente.

Creemos que las pruebas basadas en propiedades son complementarias a las pruebas unitarias: abordan cuestiones diferentes y cada una aporta sus propios beneficios. Si no las utiliza actualmente, deles una oportunidad.

Las secciones relacionadas incluyen

- Tema 23, “Diseño por contrato”.
- Tema 25, “Programación asertiva”.
- Tema 45, “El pozo de los requisitos”.

Ejercicios

Ejercicio 31

Vuelva a mirar el ejemplo del almacén. ¿Hay otras propiedades que pueda probar?

Ejercicio 32

Su empresa envía maquinaria. Cada máquina viene en una caja y todas las cajas son rectangulares. Las cajas tienen tamaños variados. Su trabajo es escribir algo de código para empaquetar tantas cajas como sea posible en una sola capa que quepa en el camión de reparto. La salida del código es una lista de todas las cajas. Para cada caja, la lista ofrece la ubicación en el camión, junto con la anchura y la altura. ¿Qué propiedades de la salida podrían probarse?

Retos

Piense en el código en el que está trabajando actualmente. ¿Cuáles son las propiedades: los contratos e invariantes? ¿Puede utilizar un *framework* de pruebas basadas en propiedades para verificarlas de manera automática?

43 Tenga cuidado ahí fuera

Las buenas vallas hacen buenos vecinos.

—Robert Frost, Mending Wall

En la explicación sobre el acoplamiento del código de la primera edición, hicimos una afirmación atrevida e ingenua: “No hace falta que seamos tan paranoicos como espías o disidentes”. Nos equivocábamos. En realidad, sí tenemos que ser así de paranoicos, todos los días.

Mientras escribimos esto, las noticias diarias están repletas de historias sobre devastadoras violaciones de datos, sistemas secuestrados y ciberfraude. Cientos de millones de registros robados de un solo golpe, miles y miles de millones de dólares en pérdidas y rectificaciones, y estas cifras crecen con rapidez año tras año. En la gran mayoría de los casos, no es porque los atacantes tengan una inteligencia superior, ni siquiera que sean remotamente competentes.

Es porque los desarrolladores fueron descuidados.

El otro 90 %

Cuando escribimos código, podemos pasar por varios ciclos de “¡funciona!” y “¿por qué no funciona?” con algún “es imposible que haya pasado eso...” ocasional.[13] Después de varias colinas y baches en esta subida, es fácil decir “¡uf, funciona!” y declarar el código acabado. Por supuesto, no está acabado todavía. Hemos acabado al 90 %, pero ahora tenemos que considerar el otro 90 %.

Lo siguiente que tenemos que hacer es analizar el código para buscar maneras en que puede fallar y añadir eso a nuestra *suite* de pruebas. Hay que plantearse cosas como pasar parámetros malos, fugas o recursos no disponibles; ese tipo de cosas.

En los viejos tiempos, esta evaluación de recursos internos podría haber sido suficiente, pero, hoy en día, es solo el principio, porque, además de los errores debidos a causas internas, también hay que considerar cómo podría estropear el sistema un actor externo de manera deliberada. Puede que ahora usted proteste: “Oh, a nadie le va a preocupar este código, no es importante, nadie sabe siquiera que existe este servidor...”. El mundo es muy grande y la mayor parte de él está conectada. No importa si es un chaval aburrido al otro lado del planeta, terrorismo de estado, bandas

criminales, espionaje industrial o incluso una expareja vengativa, todos están ahí fuera y van a por usted. El tiempo de supervivencia de un sistema sin parches y obsoleto en la red abierta se mide en minutos, o incluso menos.

La seguridad a través de la oscuridad no funciona.

Principios de seguridad básicos

Los programadores pragmáticos tenemos un nivel sano de paranoia. Sabemos que tenemos defectos y limitaciones y que los atacantes externos aprovecharán cualquier oportunidad que les demos para poner en peligro nuestros sistemas. Nuestros entornos de desarrollo y despliegue particulares tendrán sus propias necesidades centradas en la seguridad, pero también hay algunos principios básicos que debería tener en mente siempre:

1. Minimizar el área de superficie de ataque.
2. Principio de mínimo privilegio.
3. Valores predeterminados seguros.
4. Encriptar datos sensibles.
5. Mantener actualizaciones de seguridad.

Vamos a echar un vistazo a cada uno de ellos.

Minimizar el área de superficie de ataque

El área de superficie de ataque de un sistema es la suma de todos los puntos de acceso en los que un atacante puede introducir datos, extraer datos o invocar la ejecución de un servicio. Estos son algunos ejemplos:

- **La complejidad del código lleva a vectores de ataque:** La complejidad del código hace que la superficie de ataque sea más grande, con más oportunidades para efectos secundarios imprevistos. Piense en el código complejo como en hacer que el área de la superficie sea más porosa y abierta a las infecciones. Una vez más, el código simple y más pequeño es mejor. Menos código significa menos fallos, menos oportunidades para agujeros incapacitantes en la

seguridad. El código más simple, más ajustado y menos complejo es más fácil de razonar y es más fácil detectar debilidades potenciales.

- **Los datos de entrada son un vector de ataque:** Nunca confíe en datos de una entidad externa, desinféctelos siempre antes de pasarlos a una base de datos, a la representación de una vista o a otra forma de procesamiento.[14] Algunos lenguajes pueden ayudar con esto. En Ruby, por ejemplo, las variables que albergan entradas externas están "contaminadas", lo cual limita las operaciones que pueden realizarse en ellas. Por ejemplo, este código usa en apariencia la utilidad `wc` para informar del número de caracteres en un archivo cuyo nombre se proporciona en tiempo de ejecución:

safety/taint.rb

```
puts "Enter a file name to count: "
name = gets
system("wc -c #{name}")
```

Un usuario malvado podría causar daños como este:

```
Enter a file name to count:
test.dat; rm -rf /
```

Sin embargo, establecer el nivel de SAFE como 1 contaminará los datos externos, lo que significa que no pueden utilizarse en contextos peligrosos:

```
▶ $SAFE = 1

puts "Enter a file name to count: "
name = gets
system("wc -c #{name}")
```

Ahora, cuando lo ejecutamos, nos pillan con las manos en la masa:

```
$ ruby taint.rb
Enter a file name to count:
test.dat; rm -rf /
code/safety/taint.rb:5:in   `system':   Insecure   operation-system
(SecurityError)
   from code/safety/taint.rb:5:in `main
```

- **Los servicios no autenticados son un vector de ataque:** Por su propia naturaleza, cualquier usuario en cualquier parte del mundo puede llamar a servicios no autenticados, así que, salvo que haya cualquier otra gestión o limitación, ha creado de inmediato una oportunidad para un ataque de denegación de servicio, como mínimo. Bastantes filtraciones de datos muy públicas y recientes se han producido porque los desarrolladores pusieron por accidente datos en almacenes de datos no autenticados y de lectura pública en la nube.
- **Los servicios autenticados son un vector de ataque:** Mantenga el número de usuarios autorizados al mínimo absoluto. Sacrifique usuarios y servicios que no se usen, que sean antiguos o que estén desactualizados. Se han encontrado muchos dispositivos con red que contienen contraseñas predeterminadas o cuentas administrativas sin protección y que no se usan. Si una cuenta con credenciales de despliegue se ve comprometida, todo nuestro producto está en peligro.
- **Los datos de salida son un vector de ataque:** Existe una historia (posiblemente apócrifa) acerca de un sistema que informaba con diligencia del mensaje de error `La contraseña es utilizada por otro usuario`. No revele información. Asegúrese de que los datos de los que informa son apropiados para la autorización de ese usuario. Trunque u ofusque la información potencialmente peligrosa, como números de la Seguridad Social u otros números de identificación.
- **La información de la depuración es un vector de ataque:** No hay nada más reconfortante que ver un seguimiento de pila completo con los datos de nuestro cajero automático local, un quiosco en el aeropuerto o una página web que se cuelga. La información diseñada para facilitar la depuración también puede hacer que sea más fácil entrar a la fuerza. Asegúrese de que cualquier “ventana de pruebas” (consulte la sección pertinente en el tema 41) y generación de informes de excepción en tiempo de ejecución están protegidas de ojos espías.[15]

Truco 72. Manténgalo simple y minimice las superficies de ataque.

Principio de mínimo privilegio

Otro principio clave es utilizar la menor cantidad de privilegios durante el periodo de tiempo más corto que se pueda permitir. Dicho de otro modo, no elija de forma automática el nivel de permiso más alto posible, como `root` o `Administrator`. Si ese nivel alto es necesario, tómelo, haga la menor cantidad de trabajo posible y retire ese permiso enseguida para reducir el riesgo. Este principio data de principios de los setenta:

> *Todo programa y todo usuario privilegiado del sistema debería actuar utilizando la menor cantidad de privilegios necesaria para completar el trabajo.*
>
> *—Jerome Saltzer, Communications of the ACM, 1974.*

Fíjese en el programa `login` de los sistemas derivados de Unix. Inicialmente, se ejecuta con privilegios de *root*. Sin embargo, en cuanto termina de autenticar el usuario correcto, reduce el nivel de privilegios a los de usuario.

Esto no se aplica solo a los niveles de privilegios del sistema operativo. ¿Implementa su aplicación niveles diferentes de acceso? ¿Es una herramienta contundente, como “administrador” frente a “usuario”? Si es así, considere algo más fino, donde sus recursos sensibles se dividan en categorías diferentes y los usuarios individuales tengan permisos solo para algunas de esas categorías.

Esta técnica sigue el mismo tipo de idea que la minimización del área de la superficie: reducir el alcance de los vectores de ataque, tanto por tiempo como por nivel de privilegios. En este caso, menos es más, realmente.

Valores predeterminados seguros

Las opciones predeterminadas en su aplicación, o para los usuarios de su sitio, deberían ser los valores más seguros. Puede que no sean los valores más convenientes o más fáciles de usar, pero es mejor dejar que cada individuo decida por sí mismo qué le compensa más entre la seguridad y la conveniencia. Por ejemplo, la opción predeterminada para la introducción de una contraseña podría ser ocultar la contraseña a medida que va introduciéndose, reemplazando cada carácter con un asterisco. Si va a introducir la contraseña en un lugar público concurrido o la va a proyectar

ante muchas personas, es una opción predeterminada sensata, pero puede que algunos usuarios prefieran ver la contraseña deletreada, quizá por cuestiones de accesibilidad. Si no hay apenas riesgo de que alguien la lea por encima de su hombro, es una opción razonable para ellos.

Encriptar datos sensibles

No deje información personal identificable, datos financieros, contraseñas u otras credenciales en texto simple, ya sea en una base de datos o en algún otro archivo externo. Si los datos quedan expuestos, la encriptación ofrece un nivel adicional de seguridad.

En “Control de versiones” recomendábamos encarecidamente poner todo lo necesario para el proyecto en un sistema de control de versiones. Bueno, casi todo. Hay una excepción importante a esa regla:

No envíe secretos, claves API, claves SSH, contraseñas de encriptación u otras credenciales junto con su código fuente al sistema de control de versiones.

Las claves y los secretos necesitan administrarse por separado, a través de archivos de configuración o variables de entorno como parte de la construcción y el despliegue.

Mantener actualizaciones de seguridad

Actualizar sistemas informáticos puede ser una lata. Necesitamos ese parche de seguridad, pero, como efecto secundario, se estropea alguna porción de la aplicación. Podríamos decidir esperar y posponer la actualización. Es una idea terrible, porque ahora nuestro sistema es vulnerable a *exploit* conocido.

Truco 73. Aplique parches de seguridad con rapidez.

Este truco afecta a todos los dispositivos conectados a la red, incluyendo teléfonos, coches, electrodomésticos, portátiles personales, ordenadores de desarrolladores, máquinas de construcción, servidores de producción e imágenes en la nube. Todo. Y, si cree que esto no importa mucho, recuerde

que las mayores filtraciones de datos de la historia (hasta ahora) estuvieron causadas por sistemas que no tenían sus actualizaciones al día.

No deje que eso le pase a usted.

Antipatrones de contraseñas

Uno de los problemas fundamentales con la seguridad es que, a menudo, la buena seguridad va en contra del sentido común o de las prácticas habituales. Por ejemplo, puede que crea que unos requisitos de contraseña estrictos incrementarían la seguridad de su aplicación o sitio. Se equivoca.
En realidad, las políticas estrictas para las contraseñas reducirán la seguridad. Veamos una lista corta de ideas muy malas, junto con algunas recomendaciones del Instituto Nacional de Estándares y Tecnología de EE. UU. (NIST):[16]

- No restrinja la longitud de las contraseñas a menos de 64 caracteres. El NIST recomienda 256 como una buena longitud máxima.
- No trunque la contraseña elegida por el usuario.
- No restrinja los caracteres especiales, como `[]();&%$#` o `/`. Consulte la nota sobre Bobby Tables que hemos visto antes en esta sección. Si los caracteres especiales en su contraseña ponen en peligro su sistema, tiene problemas más serios. El NIST dice que hay que aceptar todos los caracteres ASCII de impresión, el espacio y Unicode.
- No proporcione indicios de contraseña a usuarios no autenticados ni solicite tipos específicos de información (por ejemplo, “¿cómo se llamaba su primera mascota?”).
- No deshabilite la función `paste` en el navegador. Inutilizar la funcionalidad del navegador y los administradores de contraseñas no hace que su sistema sea más seguro y, de hecho, lleva a los usuarios a crear contraseñas más cortas y simples que son mucho más fáciles de comprometer. Tanto el NIST en EE. UU. como el Centro Nacional de Ciberseguridad (*National Cyber Security Centre*) del Reino Unido requieren de manera específica a los verificadores que permitan la funcionalidad para pegar por esta razón.
- No imponga otras reglas de composición. Por ejemplo, no exija una mezcla concreta de mayúsculas y minúsculas, números o caracteres especiales ni prohíba repetir caracteres, etc.
- No obligue a los usuarios de manera arbitraria a cambiar sus contraseñas tras un periodo de tiempo determinado. Hágalo solo por razones válidas (por ejemplo, si ha habido una violación de la seguridad).

Le interesa incentivar el uso de contraseñas aleatorias largas con un grado alto de entropía. Poner restricciones artificiales limita la entropía y fomenta los malos hábitos respecto a las contraseñas, lo que hace que las cuentas de sus usuarios sean vulnerables a la posibilidad de que les roben el control.

Sentido común vs. criptografía

Es importante tener en cuenta que el sentido común puede fallar cuando se trata de cuestiones de criptografía. La primera regla, y la más importante, en lo que respecta a la criptografía es “nunca lo haga usted mismo”.[17] Incluso si se trata de algo tan simple como las contraseñas, las prácticas habituales están equivocadas (consulte el cuadro “Antipatrones de contraseñas”). Una vez que se entra en el mundo de la criptografía, incluso el error más pequeño que parece el más insignificante puede ponerlo todo en peligro: es probable que un experto pueda romper en minutos su nuevo e inteligente algoritmo de encriptación casero. No le conviene hacer la encriptación usted mismo.

Como ya hemos dicho antes, dependa solo de cosas fiables: *frameworks* y bibliotecas preferentemente de código abierto, bien revisados, examinados de forma exhaustiva, bien mantenidos y actualizados con frecuencia.

Más allá de las tareas de encriptación simples, fíjese bien en otras características relacionadas con la seguridad de su sitio o aplicación. Piense en la autenticación, por ejemplo.

Para implementar su propio inicio de sesión con contraseña o autenticación biométrica, necesita entender cómo funcionan los *hashes* y las sales, cómo los *crackers* usan cosas como las tablas arcoíris, por qué no debería utilizar MD5 o SHA1 y varias cuestiones más. Incluso si entiende bien todo eso, al final sigue siendo responsable de conservar los datos y mantenerlos seguros, cumpliendo cualquier legislación u obligaciones legales nuevas que surjan.

O podría adoptar el enfoque pragmático y dejar que otra persona se preocupe por eso y utilizar un proveedor de autenticación de terceros. Puede que sea un servicio existente estándar que use a nivel interno o podría ser de un tercero en la nube. A menudo, los proveedores de correo electrónico, teléfono o redes sociales tienen servicios de autenticación disponibles, que pueden ser o no ser apropiados para su aplicación. En cualquier caso, esas personas se pasan el día manteniendo sus sistemas seguros, y se les da mejor que a usted.

Tenga cuidado ahí fuera.

Las secciones relacionadas incluyen

- Tema 23, “Diseño por contrato”.
- Tema 24, “Los programas muertos no mienten”.
- Tema 25, “Programación asertiva”.
- Tema 38, “Programar por casualidad”.
- Tema 45, “El pozo de los requisitos”.

44 Poner nombre a las cosas

El principio de la sabiduría es llamar a las cosas por su nombre.

—Confucio

¿Qué hay en un nombre? Cuando estamos programando, la respuesta es: “¡Todo!”.

Creamos nombres para aplicaciones, subsistemas, módulos, funciones, variables... Estamos todo el tiempo creando cosas nuevas y poniéndoles nombres. Y esos nombres son muy muy importantes, porque revelan mucho acerca de sus intenciones y creencias.

Nosotros creemos que a las cosas se les debería poner nombre en función del papel que juegan en el código. Esto significa que, cada vez que creamos algo, tenemos que pararnos a pensar: “¿Cuál es mi motivación para crear esto?”.

Se trata de una pregunta muy importante, porque nos saca de la mentalidad de resolución de problemas inmediata y nos hace pensar en el panorama general. Cuando consideramos el papel de una variable o una función, estamos pensando en lo que tiene de especial, en lo que puede hacer y en con qué interactúa. A menudo, acabamos dándonos cuenta de que lo que estábamos a punto de hacer no tenía sentido, y todo porque no hemos podido encontrar un nombre apropiado.

Hay algo de ciencia detrás de la idea de que los nombres son muy significativos. Resulta que el cerebro puede leer y entender palabras muy deprisa: más rápido que muchas otras actividades. Eso significa que las palabras tienen cierta prioridad cuando intentamos darle sentido a algo. Esto puede demostrarse utilizando el efecto Stroop.[18]

Eche un vistazo al siguiente panel (figura 7.2). Tiene una lista de nombres de colores o tonos y cada uno se muestra en un color o tono. Pero los nombres y los colores no tienen por qué coincidir. Esta es la primera parte del reto: diga en voz alta el nombre de cada color como está escrito.[19]

Figura 7.2.

Ahora repita esto, pero esta vez diciendo en voz alta el color utilizado para dibujar la palabra. Es más difícil, ¿verdad? Es fácil hablar con fluidez cuando se lee, pero es bastante más complicado cuando intentamos reconocer colores.

Nuestro cerebro trata las palabras escritas como algo que debe respetarse. Tenemos que asegurarnos de que los nombres que utilizamos están a la altura.

Veamos un par de ejemplos:

- Estamos autenticando a personas que acceden a nuestro sitio, donde se venden joyas hechas a partir de tarjetas gráficas viejas:

```
let user = authenticate(credentials)
```

  La variable es `user` (usuario) porque siempre es `user`. Pero ¿por qué? No significa nada. ¿Qué tal `customer` (cliente) o `buyer` (comprador)? De esa manera tenemos un recordatorio constante mientras escribimos el código de lo que está intentando hacer esa persona y de lo que eso significa para nosotros.

- Tenemos un método de instancia que hace un descuento en un pedido:

```
public void deductPercent(double amount)
// ...
```

Aquí hay dos cosas. Primero, `deductPercent` es lo que hace y no por qué lo hace. Después, el nombre del parámetro `amount` es, en el mejor de los casos, confuso: ¿es una cantidad absoluta, un porcentaje? Quizá sería mejor esto:

```
public void applyDiscount(Percentage discount)
// ...
```

Ahora el nombre del método deja claro su propósito. También hemos cambiado el parámetro de `double` a `Percentage`, un tipo que hemos definido. No sabemos si a usted le pasa, pero, cuando se habla de porcentajes, nosotros nunca sabemos si se supone que el valor está entre 0 y 100 o 0,0 y 1,0. Usar un tipo documenta lo que espera la función.

- Tenemos un módulo que hace cosas interesantes con los números de Fibonacci. Una de esas cosas es calcular el número `nth` en la secuencia. Párese a pensar cómo llamaría a esta función.
  La mayoría de la gente a la que preguntamos la llamaría `fib`. Parece razonable, pero recuerde que normalmente se llama en el contexto de su módulo, así que la llamada sería `Fib.fib(n)`. ¿Qué tal si la llamamos `of` o `nth`?:

```
Fib.of(0) # => 0
Fib.nth(20) # => 4181
```

Al poner nombre a las cosas, estamos todo el tiempo buscando formas de aclarar lo que queremos decir y ese acto de clarificación nos llevará a un mejor entendimiento de nuestro código a medida que lo escribimos.

Sin embargo, no todos los nombres tienen que ser candidatos a un premio literario.

La excepción que confirma la regla

Aunque nos esforzamos por lograr la claridad en el código, la creación de marca es una cuestión totalmente diferente.

Existe una tradición arraigada que señala que los proyectos y los equipos de proyecto deberían tener nombres oscuros, “inteligentes”. Nombres de *Pokémon*, superhéroes de Marvel, mamíferos monos, personajes de *El señor de los anillos*, o como quiera llamarlos.
Literalmente.

Respetar la cultura

Solo hay dos cosas difíciles en la informática: la invalidación de la caché y poner nombre a las cosas.

La mayoría de los textos de introducción a la informática le advertirán que nunca debe utilizar una sola letra para variables, como `i`, `j` o `k`.[20]

Nosotros creemos que se equivocan. Más o menos.

De hecho, depende de la cultura de ese entorno o lenguaje de programación en particular. En el lenguaje de programación C, `i`, `j` y `k` se utilizan tradicionalmente como variables de incremento del bucle, `s` se utiliza para una cadena de caracteres, etc. Si programa en ese entorno, eso es lo que está acostumbrado a ver y sería discordante (y, por tanto, equivocado) violar esa norma. Por otra parte, usar esa convención en un entorno diferente donde no es lo que se espera es un error. Nunca hará algo atroz como este ejemplo de Clojure que asigna una cadena a la variable `i`:

```
(let [i “Hello World”]
(println i))
```

Algunas comunidades de lenguajes prefieren `camelCase`, con letras mayúsculas incrustadas, mientras que otras prefieren `snake_case` con guiones bajos incrustados para separar palabras. Por supuesto, los lenguajes en sí aceptarán cualquiera de las dos opciones, pero eso no quiere decir que esté bien. Respete la cultura local.

Algunos lenguajes permiten un subconjunto de Unicode en los nombres. Hágase una idea de lo que espera la comunidad antes de lanzarse a poner nombres bonitos como ɹəsɲ o εξέρΧεται

Coherencia

Emerson es famoso por escribir: “Una consistencia tonta es el duende de las mentes pequeñas...”, pero Emerson no estaba en un equipo de programadores.

Cada proyecto tiene su propio vocabulario: las palabras de una jerga que tienen un significado especial para el equipo. “Order” (que en este contexto se traduce como “pedido”) significa una cosa para un equipo que está creando una tienda en línea, y algo muy diferente para un equipo cuya aplicación traza el linaje de grupos religiosos (en este contexto se traduce como “orden”). Es importante que todos los miembros del equipo sepan qué significan esas palabras y las usen de manera coherente.

Una forma es fomentar la comunicación. Si todo el mundo programa en pareja y las parejas se cambian con frecuencia, la jerga se extenderá por ósmosis.

Otra manera es contar con un glosario del proyecto, con una lista de los términos que tienen un significado especial para el equipo. Se trata de un documento informal, que puede mantenerse en una wiki o ser solo unas fichas clavadas en una pared en algún sitio.

Después de un tiempo, la jerga del proyecto cobrará vida propia. Cuando todo el mundo esté cómodo con el vocabulario, se podrá utilizar la jerga como taquigrafía, expresando una gran cantidad de significado de manera precisa y concisa. (Eso es exactamente un lenguaje de patrones).

Cambiar nombres es aún más difícil

> *Hay dos problemas difíciles en la informática: la invalidación de la caché, poner nombre a las cosas y los errores por uno.*

No importa cuánto nos esforcemos de antemano, las cosas cambian. El código se refactoriza, el uso cambia, el significado se altera con sutileza. Si no se preocupa por la actualización de los nombres a medida que avanza, puede que acabe enseguida atrapado en una pesadilla mucho peor que los nombres sin significado: los nombres engañosos. ¿Alguna vez ha oído a alguien explicar incoherencias en el código como “La rutina llamada `getData` en realidad escribe datos en un archivo de almacenamiento”?

Como decíamos en “Entropía del software”, cuando detecte un problema, arréglelo en ese mismo momento. Cuando vea un nombre que ya

no expresa el propósito o que es engañoso o confuso, arréglelo. Tiene pruebas de regresión completas, así que detectará cualquier caso que se le haya pasado por alto.

Truco 74. Ponga bien los nombres; cámbielos cuando sea necesario.

Si, por alguna razón, no puede cambiar el nombre que ahora está mal, entonces tiene un problema mayor: la violación del principio ETC (consulte "La esencia del buen diseño"). Solucione eso primero y, después, cambie el nombre problemático. Haga que cambiar los nombres sea fácil y hágalo a menudo.

De lo contrario, tendrá que explicar a los nuevos miembros del equipo que, en realidad, `getData` escribe datos en un archivo, y tendrá que hacerlo con un rostro impasible.

Las secciones relacionadas incluyen

- Tema 3, "Entropía del software".
- Tema 40, "Refactorización".
- Tema 45, "El pozo de los requisitos".

Retos

- Cuando encuentre una función o un método con un nombre demasiado genérico, pruebe a cambiárselo para que exprese todas las cosas que hace en realidad. Así será un objetivo más fácil para la refactorización.
- En nuestros ejemplos, sugeríamos utilizar nombres más específicos, como "comprador" en vez del tradicional y genérico "usuario". ¿Qué otros nombres ve con frecuencia que podrían cambiar por otros mejores?
- ¿Son los nombres en su sistema congruentes con los términos de usuario del dominio? Si no es así, ¿por qué no? ¿Provoca esto una disonancia cognitiva del estilo del efecto Stroop en el equipo?

- ¿Hay nombres en su sistema difíciles de cambiar? ¿Qué puede hacer para arreglar esa ventana rota concreta?

---

[1] Nota de los escarmentados: el UTC existe por una razón. Úselo.

[2] https://es.wikipedia.org/wiki/Cum_hoc_ergo_propter_hoc.

[3] Consulte el tema 50, "Los cocos no sirven".

[4] Aquí también puede ir demasiado lejos. Una vez conocimos a un desarrollador que reescribió toda la fuente que se le dio porque tenía sus propias convenciones para los nombres.

[5] https://media-origin.pragprog.com/titles/tpp20/code/algorithm_speed/sort/src/main.rs.

[6] Y sí, expresamos nuestra preocupación por el título.

[7] Se encuentran por primera vez en *UML Distilled: A Brief Guide to the Standard Object Modeling Language* [Fow00].

[8] Este es un consejo excelente, en general (consulte el tema 27, "No vaya más rápido que sus faros").

[9] Hay quienes argumentan que escribir las pruebas primero y el desarrollo guiado por pruebas son dos cosas diferentes, diciendo que los propósitos de ambas son distintos. Sin embargo, desde el punto de vista histórico, escribir las pruebas primero (que viene de la Programación Extrema) era idéntico a lo que la gente denomina hoy TDD.

[10] https://ronjeffries.com/categories/sudoku. Muchas gracias a Ron por dejarnos usar esta historia.

[11] http://norvig.com/sudoku.html.

[12] Estamos intentándolo al menos desde 1986, cuando Cox y Novobilski acuñaron el término "Software-IC" (que se puede traducir como "circuito integrado de software") en su libro sobre Objective-C *Programación orientada a objetos: un enfoque evolutivo* [CN91].

[13] Consulte "Depuración".

[14] ¿Recuerda a nuestro buen amigo, el pequeño Bobby Tables (https://xkcd.com/327)? Mientras se acuerda, eche un vistazo a https://bobby-tables.com, que ofrece una lista de maneras de desinfectar los datos pasados a las consultas de la base de datos.

15 Esta técnica ha demostrado tener éxito a nivel del chip de la CPU, donde los conocidos *exploits* se dirigen a instalaciones de depuración y administrativas. Una vez craqueada, la máquina al completo queda expuesta.

16 *NIST Special Publication 800-63B: Digital Identity Guidelines: Authentication and Lifecycle Management*, disponible en línea de forma gratuita en https://doi.org/10.6028/NIST.SP.800-63b.

17 A menos que tenga un doctorado en criptografía, e incluso en ese caso, solo con una revisión por pares importante, ensayos de campo completos con recompensas por errores y presupuesto para un mantenimiento a largo plazo.

18 *Studies of Interference in Serial Verbal Reactions* [Str35].

19 Tenemos dos versiones de este panel. Una utiliza colores diferentes y la otra utiliza tonos de gris. Si está viendo esto en la versión en blanco y negro y quiere la versión en color o si tiene problemas para distinguir los colores y quiere probar la versión en escala de grises, encontrará estas imágenes en la página web del libro.

20 ¿Sabe por qué la `i` se usa de forma habitual como variable de bucle? La respuesta viene de hace 60 años, cuando las variables que empezaban con letras de la `I` a la `N` eran enteros en el FORTRAN original. Y, a su vez, FORTRAN estaba influido por el álgebra.

8
Antes del proyecto

Al principio de un proyecto, usted y su equipo necesitan conocer los requisitos. Que les digan solo qué hacer o escuchar a los usuarios sin más no es suficiente: lea “El pozo de los requisitos” y aprenda cómo evitar las trampas y los obstáculos habituales.

La sabiduría popular y la gestión de restricciones son los temas de “Resolver rompecabezas imposibles”. No importa si está ocupándose de los requisitos, el análisis, la escritura del código o las pruebas, aparecerán problemas difíciles de la nada. La mayor parte del tiempo, no serán tan difíciles como parecen.

Y, cuando surge ese proyecto imposible, nos gusta sacar nuestra arma secreta: “Trabajar juntos”. Y con “trabajar juntos” no nos referimos a compartir documentos de requisitos masivos, enviar correos electrónicos con un montón de gente en copia o soportar reuniones eternas. Nos referimos a resolver problemas juntos mientras escribimos código. Le mostraremos a quién necesita y cómo empezar.

Aunque el Manifiesto Ágil empieza con “Individuos e interacciones sobre procesos y herramientas”, prácticamente todos los proyectos “ágiles” comienzan con una discusión irónica acerca de qué procesos y qué herramientas se van a utilizar. Pero no importa lo bien planeado que esté, y al margen de qué “buenas prácticas” incluya, ningún proyecto puede sustituir al pensamiento. No necesita ningún proyecto o herramienta en particular, lo que necesita es “La esencia de la agilidad”.

Con estas cuestiones cruciales resueltas antes de que el proyecto se ponga en marcha, puede estar mejor posicionado para evitar la “parálisis del análisis” y comenzar (y completar) de verdad su proyecto con éxito.

45 El pozo de los requisitos

La perfección se consigue, no cuando no hay más que añadir, sino cuando no queda nada más por quitar...

—Antoine de St. Exupéry, Viento, arena y estrellas, 1939

Muchos libros y tutoriales se refieren a la recopilación de requisitos como una frase temprana del proyecto. La palabra “recopilación” parece implicar a un grupo de analistas felices rebuscando pepitas de conocimiento que están por el suelo a su alrededor mientras suena la Sinfonía Pastoral de fondo. “Recopilar” implica que los requisitos ya están ahí y solo tenemos que encontrarlos, meterlos en la cesta y marcharnos tan contentos.

No funciona así. Los requisitos rara vez están en la superficie. Por lo general, están enterrados bajo capas de asunciones, ideas equivocadas y políticas. Lo que es peor, a menudo ni siquiera existen.

Truco 75. Nadie sabe con exactitud lo que quiere.

El mito de los requisitos

En los primeros días del software, los ordenadores eran más valiosos (en lo relativo al coste por hora amortizado) que las personas que trabajaban con ellos. Ahorrábamos dinero intentando hacer las cosas bien a la primera. Parte de ese proceso era intentar especificar con exactitud lo que íbamos a hacer que hiciese la máquina. Empezábamos por obtener una especificación de los requisitos, la transformábamos en un documento de diseño, después en diagramas de flujo y pseudocódigo y, por último, en código. Sin embargo, antes de introducirlo en un ordenador, dedicábamos tiempo a comprobarlo a mano.

Costaba mucho dinero, y ese coste significaba que la gente solo intentaba automatizar algo cuando sabía exactamente lo que quería. Como los primeros ordenadores eran bastante limitados, el ámbito de los problemas que resolvían estaba restringido: en realidad, era posible entender el problema completo antes de empezar.

Pero el mundo real no es así. Es desordenado, en conflicto y desconocido. En ese mundo, las especificaciones exactas de cualquier cosa son muy poco comunes, por no decir totalmente imposibles.

Ahí es donde entramos los programadores. Nuestro trabajo es ayudar a la gente a entender lo que quiere. De hecho, es probable que ese sea nuestro atributo más importante, así que vale la pena repetirlo:

Truco 76. Los programadores ayudan a la gente a entender lo que quiere.

Programar como terapia

Vamos a denominar a las personas que nos piden que escribamos software nuestros clientes. El cliente típico acude a nosotros con una necesidad. Esa necesidad puede ser estratégica, pero es igual de probable que sea una cuestión táctica: una respuesta a un problema actual. La necesidad puede ser un cambio en un sistema existente o algo nuevo. A veces se expresará con términos empresariales y otras veces con términos técnicos.

El error que cometen con frecuencia los desarrolladores nuevos es que toman esa declaración de la necesidad e implementan una solución para ella.

Según nuestra experiencia, esa declaración inicial de la necesidad no es un requisito absoluto. Puede que el cliente no se dé cuenta de esto, pero en realidad es una invitación a la exploración.

Vamos a ver un ejemplo sencillo. Trabaja para una editorial de libros en papel y electrónicos y se le da un nuevo requisito:

> *El envío debería ser gratuito para todos los pedidos de 50 dólares o superiores.*

Pare un momento e imagínese en esa posición. ¿Qué es lo primero que le viene a la cabeza? Hay muchas probabilidades de que tenga preguntas:

- ¿Los 50 dólares incluyen impuestos?
- ¿Los 50 dólares incluyen los gastos de envío actuales?
- ¿Tienen que ser los 50 dólares para libros en papel o el pedido puede incluir también libros electrónicos?
- ¿Qué tipo de envío se ofrece? ¿Prioritario? ¿Terrestre?
- ¿Qué pasa con los pedidos internacionales?
- ¿Con qué frecuencia cambiará el límite de 50 dólares en el futuro?

Eso es lo que hacemos. Cuando se nos da algo que parece simple, molestamos a la gente buscando casos límite y preguntando al respecto.

Lo más probable es que el cliente ya haya pensado en algunas de estas cosas y que haya asumido sin más que la implementación funcionaría así. Plantear la pregunta simplemente hace que esa información salga a la luz.

Pero es posible que haya otras preguntas que sean cuestiones que el cliente no se ha planteado con anterioridad. Aquí es donde la cosa se pone interesante y donde un buen desarrollador aprende a ser diplomático.

> ***Usted:*** *Nos estábamos preguntando por esos 50 dólares en total. ¿Se incluye ahí lo que cobraríamos normalmente por el envío?*
> ***Cliente:*** *Sí, claro. Es el total que nos pagarían.*
> ***Usted:*** *Eso está bien y es fácil que los compradores lo entiendan, veo el atractivo. Pero también veo a gente con menos escrúpulos intentando jugar con ese sistema.*
> *Cliente: ¿Cómo?*
> ***Usted:*** *Bueno, digamos que compran un libro por 25 dólares y después seleccionan el envío con entrega al día siguiente, que es la opción más cara. Eso podrían ser unos 30 dólares, lo que haría que el pedido al final fuese de 55 dólares. Entonces aplicaríamos el envío gratuito y recibirían de un día para otro un libro de 25 dólares por solo 25 dólares.*
> *(En este punto, el desarrollador experimentado se calla. Ha expuesto los hechos y deja que el cliente tome las decisiones).*
> ***Cliente:*** *Buf. Vale, eso no es lo que tenía pensado. Perderíamos mucho dinero con esos pedidos. ¿Cuáles son las opciones?*

Y aquí es donde empieza la exploración. Su función aquí es interpretar lo que dice el cliente y explicarle las implicaciones. Este proceso es tanto intelectual como creativo: está pensando con rapidez y está contribuyendo a una solución que tiene probabilidades de ser mejor que la que usted o su cliente habrían producido solos.

Los requisitos son un proceso

En el ejemplo anterior, el desarrollador tomó los requisitos y explicó una consecuencia al cliente. Eso inició la exploración. Durante esa exploración, es probable que pueda aportar más opiniones a medida que el cliente juega con diferentes soluciones. Esta es la realidad de la recopilación de requisitos:

Truco 77. Los requisitos se aprenden en un bucle de *feedback*.

Su trabajo es ayudar al cliente a entender las consecuencias de sus requisitos solicitados. Para hacerlo, debe generar *feedback* y dejarle utilizar dicho *feedback* para perfeccionar su razonamiento.

En el ejemplo anterior, el *feedback* podía expresarse en palabras con facilidad. A veces, no es así. Y, a veces, no sabrá lo bastante acerca del dominio para ser así de específico.

En esos casos, los programadores pragmáticos dependemos de la escuela de *feedback* "¿es esto lo que quiere decir?". Producimos modelos y prototipos y dejamos que el cliente juegue con ellos. Lo ideal es que las cosas que produzcamos sean lo bastante flexibles para que podamos cambiarlas durante nuestras conversaciones con el cliente, dejándonos la posibilidad de responder a "eso no es lo que quería decir" con "entonces, ¿algo más parecido a esto?".

A veces, estos modelos pueden hacerse en una hora, más o menos. Está claro que solo son ayudas para transmitir bien una idea.

Pero la realidad es que todo el trabajo que hacemos es algún tipo de modelo. Incluso al final de un proyecto, seguimos interpretando lo que quiere nuestro cliente. De hecho, para cuando llegue ese momento es probable que ya tengamos más clientes: el personal de QA, el de operaciones, el de marketing y, quizá, incluso grupos de prueba de consumidores.

Así pues, un programador pragmático ve todo el proyecto como un ejercicio de recopilación de requisitos. Por eso preferimos las iteraciones cortas, que acaben con *feedback* directo del cliente. Eso nos mantiene en el camino correcto y garantiza que, si vamos en la dirección equivocada, la cantidad de tiempo que perdemos es mínima.

Póngase en la piel del cliente

Hay una técnica simple para meternos en la cabeza del cliente que no se utiliza con la suficiente frecuencia: conviértase en un cliente. ¿Está escribiendo un sistema para el servicio técnico? Pase un par de días monitorizando los teléfonos con una persona del servicio experimentada. ¿Está automatizando un sistema de control de *stock* manual? Trabaje en el almacén una semana.[1]

Además de conseguir una perspectiva más profunda de cómo va a utilizarse de verdad el sistema, le sorprenderá cómo decirle a alguien "¿te importa si estoy contigo una semana mientras trabajas?" ayuda a desarrollar la confianza y establece una base para la comunicación con sus clientes. ¡Solo tenga cuidado de no estorbar!

Truco 78. Trabaje con un usuario para pensar como un usuario.

La recogida de *feedback* también es el momento de empezar a construir una buena relación con su base de clientes, enterándose de cuáles son sus expectativas y esperanzas para el sistema que está construyendo. Consulte el tema 52, "Deleite a sus usuarios", para obtener más información.

Requisitos frente a política

Supongamos que mientras hablamos de un sistema para Recursos Humanos, un cliente dice: "Solo los supervisores de un empleado y el departamento de personal pueden ver los registros de ese empleado". ¿Es esa afirmación realmente un requisito? Puede que hoy sí, pero incorpora la política de la empresa en una declaración absoluta.

¿Política de la empresa? ¿Requisito? Se trata de una distinción relativamente sutil, pero que tiene implicaciones profundas para los desarrolladores. Si el requisito se expresa como "Solo los supervisores y el personal pueden ver el registro de un empleado", puede que el desarrollador acabe escribiendo código para una prueba explícita cada vez que la aplicación acceda a estos datos. Sin embargo, si la afirmación es "Solo los usuarios autorizados pueden acceder al registro de un empleado", es probable que el desarrollador diseñe e implemente algún tipo de sistema de control de acceso. Cuando la política cambia (cosa que sucederá), solo habrá que actualizar los metadatos para ese sistema. De hecho, recopilar requisitos de este modo nos lleva, de forma natural, a un sistema que está bien programado para soportar metadatos.

Aquí hay una regla general:

Truco 79. Las políticas son metadatos.

Implemente el caso general, con la información de la política como un ejemplo del tipo de cosas que el sistema necesita soportar.

Requisitos frente a realidad

En un artículo de la revista *Wired* en enero de 1999,[2] el productor y músico Brian Eno describía una tecnología increíble: la mesa de mezclas definitiva. Hace al sonido cualquier cosa que se le pueda hacer y, aun así, en vez de permitir a los músicos crear música mejor o producir una grabación de manera más rápida o menos cara, es un estorbo; interrumpe el proceso creativo. Para ver el porqué, tenemos que fijarnos en cómo trabajan los ingenieros de grabación. Equilibran los sonidos de manera intuitiva. Con el paso de los años, desarrollan un bucle de *feedback* innato entre sus oídos y sus dedos: deslizan reguladores, giran perillas, etc. Sin embargo, la interfaz de la nueva mesa no aprovechaba esas habilidades, sino que obligaba a sus usuarios a escribir en un teclado o hacer clic con un ratón. Las funciones que ofrecía eran muy completas, pero se presentaban de maneras que resultaban exóticas y poco familiares. Las funciones que necesitaban los ingenieros estaban a veces ocultas tras nombres confusos o se lograban mediante combinaciones poco intuitivas de herramientas básicas.

Este ejemplo también ilustra nuestra creencia de que las herramientas que tienen éxito se adaptan a las manos de quien las usa. Una recopilación de requisitos que funciona tiene eso en cuenta. Y esa es la razón por la que el *feedback* temprano, con prototipos y balas trazadoras, permitirá a los clientes decir: “Sí, hace lo que quiero, pero no como quiero”.

Documentar requisitos

Nosotros consideramos que la mejor documentación de los requisitos, quizá la única documentación de los requisitos, es el código que funciona.

Pero eso no significa que pueda prescindir de documentar su entendimiento de lo que quiere el cliente. Solo significa que esos documentos no son entregables: no son algo que demos al cliente para que

le dé el visto bueno, sino que son solo hitos para ayudar a guiar el proceso de implementación.

Los documentos de requisitos no son para los clientes

En el pasado, tanto Andy como Dave hemos estado en proyectos que producían requisitos con un nivel de detalle increíble. Estos documentos trascendentales ampliaban las explicaciones iniciales de dos minutos del cliente de lo que quería, produciendo obras maestras de dos centímetros y medio de grosor llenas de diagramas y tablas. Las cosas se especificaban hasta tal punto que no había casi espacio para la ambigüedad en la implementación. Con unas herramientas lo bastante potentes, lo cierto es que el documento podría ser el programa definitivo.

Crear esos documentos fue un error por dos razones. En primer lugar, como ya hemos visto, en realidad el cliente no sabe lo que quiere por adelantado, así que, cuando cogemos lo que dice y lo ampliamos hasta convertirlo en un documento casi legal, estamos creando un castillo increíblemente complejo sobre arenas movedizas.

Puede que diga: “Pero después llevamos el documento al cliente y le da el visto bueno. Estamos recibiendo *feedback*”. Y eso nos lleva al segundo problema con estas especificaciones de los requisitos: el cliente nunca las lee.

El cliente usa a los programadores porque, mientras él tiene la motivación de resolver problemas de alto nivel y, en cierto modo, difusos, los programadores están interesados en todos los detalles y matices. El documento de los requisitos se escribe para desarrolladores y contiene información y sutilezas que a veces son incomprensibles y, a menudo, aburridas para el cliente.

Entregue un documento de requisitos de 200 páginas y lo más probable es que el cliente lo levante para decidir si pesa lo suficiente para ser importante, puede que lea los dos primeros párrafos (razón por la cual los dos primeros párrafos siempre llevan un título como “Resumen para la dirección”) y hojee el resto, parando de vez en cuando si hay algún diagrama bonito. Esto no es menospreciar al cliente, pero darle un documento técnico largo es como darle a un desarrollador medio una copia

de la Ilíada en griego homérico y pedirle que escriba código para un videojuego a partir de ella.

Los documentos de requisitos sirven para planificar

Vale, no creemos en los documentos de requisitos monolíticos y más pesados que una vaca en brazos, pero sí creemos que los requisitos deben escribirse, simplemente porque los desarrolladores de un equipo necesitan saber qué van a estar haciendo.

¿Qué forma adopta esto? Preferimos algo que quepa en una ficha real (o virtual). Estas descripciones cortas se denominan a menudo historias de usuario. Describen lo que debería hacer una pequeña porción de la aplicación desde la perspectiva de un usuario de esa funcionalidad. Cuando se escriben de este modo, los requisitos pueden ponerse en un tablón e ir moviéndose para mostrar tanto el estado como la prioridad.

Puede que le parezca que una sola ficha no puede albergar toda la información necesaria para implementar un componente de la aplicación. Tendrá razón. Y eso es parte de la cuestión. Al mantener esta declaración de requisitos breve, animamos a los desarrolladores a plantear preguntas clarificadoras. Estaremos mejorando el proceso de *feedback* entre clientes y programadores antes de y durante la creación de cada porción del código.

Sobreespecificación

Otro gran peligro en la producción de documentos de requisitos es ser demasiado específico. Los buenos requisitos son abstractos. En lo que respecta a los requisitos, lo mejor es utilizar afirmaciones simples que reflejen con exactitud la necesidad empresarial. Eso no significa que pueda ser impreciso; debe captar las invariantes semánticas subyacentes como requisitos y documentar las prácticas de trabajo específicas o actuales como política. Los requisitos no son arquitectura. No son diseño ni son la interfaz de usuario. Los requisitos son una necesidad.

Solo una cosita más...

Los fracasos de muchos proyectos se achacan a una expansión del alcance, también conocida como *feature bloat*, *creeping featurism* o *requirements creep*. Este es un aspecto del síndrome de la rana hervida mencionado en el tema 4, “Sopa de piedras y ranas hervidas”. ¿Qué podemos hacer para evitar que los requisitos se apoderen de nosotros?

La respuesta (una vez más) es el *feedback*. Si estamos trabajando con el cliente en iteraciones con un *feedback* constante, el cliente experimentará en sus propias carnes el impacto de “solo una característica más”. Verá cómo se pone otra ficha en el tablón y ayudará a elegir otra ficha para pasarla a la siguiente iteración para hacer sitio. El *feedback* va en dos direcciones.

Mantenga un glosario

En cuanto se empiece a hablar de requisitos, los usuarios y los expertos del dominio utilizarán términos determinados que tienen un significado específico para ellos. Puede que diferencien entre un “cliente” y un “consumidor”, por ejemplo. Sería poco apropiado utilizar cualquiera de las dos palabras de manera indiferente en el sistema.

Cree y mantenga un glosario del proyecto, un lugar que defina todos los términos y vocabulario específicos usados en un proyecto. Todos los participantes del proyecto, desde los usuarios finales al personal del servicio técnico, deberían utilizar el glosario para garantizar la consistencia. Esto implica que el glosario debe ser muy accesible, un buen argumento para la documentación en línea.

Truco 80. Use un glosario del proyecto.

Es difícil que el proyecto salga bien si los usuarios y los desarrolladores llaman a la misma cosa por distinto nombre o, peor aún, se refieren a cosas diferentes usando el mismo nombre.

Las secciones relacionadas incluyen

- Tema 5, “Software lo bastante bueno”.
- Tema 7, “¡Comuníquese!”.

- Tema 11, “Reversibilidad”.
- Tema 13, “Prototipos y notas en *post-its*”.
- Tema 23, “Diseño por contrato”.
- Tema 43, “Tenga cuidado ahí fuera”.
- Tema 44, “Poner nombre a las cosas”.
- Tema 46, “Resolver rompecabezas imposibles”.
- Tema 52, “Deleite a sus usuarios”.

Ejercicios

Ejercicio 33

¿Cuáles de los siguientes son probablemente requisitos auténticos? Reformule aquellos que no lo sean para hacer que resulten más útiles (si es posible).

1. El tiempo de respuesta debe ser inferior a ~500 ms.
2. Las ventanas modales tendrán un fondo gris.
3. La aplicación se organizará como varios procesos *front end* y un servidor *back end*.
4. Si un usuario introduce caracteres no numéricos en un campo numérico, el sistema hará parpadear el fondo del campo y no los aceptará.
5. El código y los datos para esta aplicación integrada deben caber en 32 Mb.

Retos

- ¿Puede utilizar el software que está escribiendo? ¿Es posible tener una idea clara de los requisitos sin ser capaz de usar el software usted mismo?
- Elija un problema no relacionado con la informática que necesite resolver en este momento. Genere requisitos para una solución no informatizada.

46 Resolver rompecabezas imposibles

Gordias, el rey de Frigia, ató una vez un nudo que nadie podía desatar. Se decía que quien resolviese el acertijo del nudo gordiano gobernaría toda Asia. Entonces llegó Alejando Magno, que cortó el nudo en pedazos con su espada. Fue solo una interpretación un poco diferente de los requisitos, eso es todo... Y acabó gobernando la mayor parte de Asia.

De vez en cuando, nos vemos enredados en mitad de un proyecto cuando surge un rompecabezas realmente difícil: una parte de ingeniería a la que no le cogemos el truco o quizá algo de código que está resultando mucho más difícil de escribir de lo que pensábamos. Puede que parezca imposible. Pero ¿es en realidad tan difícil como parece? Piense en los rompecabezas del mundo real, esas enrevesadas piezas de madera, hierro forjado o plástico que aparecen como regalos de Navidad o en mercadillos de segunda mano. Lo único que hay que hacer es sacar la anilla, o hacer que las piezas en forma de T entren en la caja, o lo que sea.

Así pues, tiramos de la anilla o metemos las T en la caja y enseguida nos damos cuenta de que las soluciones obvias no funcionan. El rompecabezas no puede resolverse así. Pero, incluso aunque sea obvio, eso no evita que la gente haga lo mismo (una y otra vez) pensando que tiene que haber una forma.

Por supuesto, no la hay. La solución está en otra parte. El secreto para resolver el rompecabezas es identificar las restricciones reales (no imaginarias) y encontrar ahí una solución. Algunas restricciones son absolutas; otras no son más que nociones preconcebidas. Las restricciones absolutas deben respetarse, no importa lo desagradables o estúpidas que parezcan.

Por otra parte, como demostró Alejandro, algunas restricciones aparentes pueden no ser restricciones reales en absoluto. Muchos problemas de software pueden ser igual de engañosos.

Grados de libertad

En inglés, existe la expresión de moda popular “*thinking outside the box*” (literalmente, “pensar fuera de la caja” para referirse a pensar de manera no convencional) que nos anima a reconocer las restricciones que

podrían no ser aplicables e ignorarlas. Pero esta expresión no es del todo exacta. Si “la caja” es el límite de las restricciones y las condiciones, entonces el truco está en encontrar la caja, que puede ser bastante más grande de lo que creemos.

La clave para resolver rompecabezas está tanto en reconocer las restricciones impuestas como en reconocer los grados de libertad que tenemos, puesto que es ahí donde encontraremos la solución. Esa es la razón por que algunos rompecabezas son tan efectivos; puede que descartemos soluciones potenciales con demasiada facilidad.

Por ejemplo, ¿puede conectar todos los puntos del siguiente rompecabezas y volver al punto inicial con solo tres líneas rectas, sin levantar el bolígrafo del papel y sin volver sobre líneas que ya ha trazado (*Math Puzzles & Games* [Hol92])?

Figura 8.1.

Debe cuestionar todas las nociones preconcebidas y evaluar si son o no restricciones reales y duras.

Solo es una cuestión de pensar dentro o fuera de la caja. El problema está en encontrar la caja, en identificar las restricciones reales.

Truco 81. No piense fuera de la caja; encuentre la caja.

Cuando se enfrente a un problema inextricable, enumere todas las vías posibles que hay ante usted. No descarte nada, por inservible o estúpido que suene. Después, repase la lista y explique por qué no puede tomarse determinada ruta. ¿Está seguro? ¿Puede demostrarlo?

Piense en el caballo de Troya: una solución novedosa para un problema muy complejo. ¿Cómo metemos a los soldados en una ciudad amurallada

sin ser descubiertos? Seguro que “por la puerta principal” se descartó al principio por ser un suicidio.

Categorice y priorice sus restricciones. Cuando los carpinteros comienzan un proyecto, cortan primero las piezas más largas y, después, sacan las más pequeñas de la madera restante. Del mismo modo, queremos identificar primero las limitaciones más restrictivas y encajar las restantes dentro de ellas.

Por cierto, la solución al rompecabezas de los cuatro puntos está al final del libro, después de las soluciones de los ejercicios.

¡No se estorbe a sí mismo!

A veces, se encontrará trabajando en un problema que parece mucho más difícil de lo que debería ser. Quizá tenga la impresión de que va por el camino equivocado; ¡tiene que haber una manera más fácil! Puede que ahora vaya con retraso respecto a los plazos programados o que esté desesperado por conseguir que el sistema funcione porque este problema en particular es “imposible”.

Este es el momento ideal para hacer otra cosa durante un rato. Trabaje en algo diferente. Pasee al perro. Échese una siesta.

Su cerebro consciente está al tanto del problema, pero su cerebro consciente es bastante tonto (no se ofenda), así que es hora de dar algo de su espacio a su cerebro real, esa asombrosa red neuronal asociativa que acecha bajo nuestra consciencia. Le sorprenderá la frecuencia con la que la respuesta aparecerá en su cabeza sin más mientras se distrae de manera intencionada.

Aunque esto le suene demasiado místico, no lo es. *Psychology Today*[3] publica:

> *Por decirlo llanamente, las personas que se distrajeron tuvieron mejores resultados en una tarea compleja de resolución de problemas que aquellas que dedicaron un esfuerzo consciente.*

Si aun así no está dispuesto a olvidarse del problema durante un rato, es probable que la siguiente mejor opción sea encontrar a alguien para explicárselo. A menudo, la distracción de hablar sobre ello sin más le proporcionará la claridad que necesita.

Haga que le planteen preguntas como:

- ¿Por qué está resolviendo este problema?
- ¿Cuál es el beneficio de resolverlo?
- ¿Están los problemas que está teniendo relacionados con casos límite? ¿Puede eliminarlos?
- ¿Hay algún problema relacionado más simple que pueda resolver?

Este es otro ejemplo de la técnica del patito de goma en la práctica.

La suerte favorece a la mente preparada

A Louis Pasteur se le atribuye la frase:

> *Dans les champs de l'observation le hasard ne favorise que les esprits préparés. (En el campo de la observación, la suerte favorece solo a las mentes preparadas).*

Esto también es cierto cuando se habla de la resolución de problemas. Para poder tener esos momentos de ¡eureka!, nuestro cerebro no consciente necesita tener materia prima de sobra; las experiencias anteriores pueden contribuir a la respuesta.

Una manera genial de alimentar a nuestro cerebro es proporcionarle *feedback* acerca de lo que funciona y lo que no en nuestro trabajo diario. Y describimos una forma estupenda de hacerlo usando un cuaderno de bitácora de ingeniería (tema 22, "Cuadernos de bitácora de ingeniería"). Y recuerde siempre el consejo de la portada de *Guía del autoestopista galáctico*: *DON'T PANIC* (Que no cunda el pánico).

Las secciones relacionadas incluyen

- Tema 5, "Software lo bastante bueno".
- Tema 37, "Escuche a su cerebro reptiliano".
- Tema 45, "El pozo de los requisitos".
- Andy escribió un libro entero acerca de este tipo de cosas: *Pragmatic Thinking and Learning: Refactor Your Wetware* [Hun08].

Retos

- Fíjese en cualquier problema difícil con el que tenga que lidiar hoy. ¿Puede cortar el nudo gordiano? ¿Tiene que hacerlo así? ¿Tiene que hacerlo siquiera?
- ¿Se le dio un conjunto de restricciones cuando aceptó su proyecto actual? ¿Son todavía aplicables y sigue siendo válida su interpretación?

47 Trabajar juntos

> *Jamás he conocido a un ser humano que quisiese leer 17.000 páginas de documentación y, si lo hubiera, lo mataría para sacarlo del acervo génico.*
>
> *—Joseph Costello, presidente de Cadence*

Era uno de esos proyectos imposibles, de esos de los que se oye hablar, que suenan emocionantes y aterradores al mismo tiempo. Un sistema antiguo estaba llegando al fin de su vida, el hardware estaba desapareciendo físicamente y había que crear un sistema completamente nuevo que se ajustase con exactitud al comportamiento (a menudo no documentado). Muchos cientos de millones de dólares de otras personas pasaban por ese sistema y el plazo desde el comienzo hasta el despliegue era de meses.

Y ahí fue donde nosotros nos conocimos. Un proyecto imposible con un plazo ridículo. Solo hubo una cosa que hizo que el proyecto fuese un éxito rotundo. La experta que había administrado este sistema durante años estaba justo ahí, en su oficina, enfrente de nuestra sala de desarrollo del tamaño de un armario escobero, siempre disponible para responder preguntas, hacer aclaraciones, tomar decisiones y asistir a demostraciones.

A lo largo de este libro, recomendamos trabajar en cercanía con los usuarios; son parte de nuestro equipo. En ese primer proyecto, practicamos lo que ahora se denominaría programación en pareja o programación en grupo: una persona teclea el código mientras otro u otros miembros del equipo comentan, sopesan y resuelven problemas juntos. Es una manera muy potente de trabajar juntos que trasciende las reuniones eternas, los memorandos y la documentación sobrecargada de aspectos legales valorada por su peso más que por su utilidad.

Y eso es a lo que nos referimos en realidad cuando decimos “trabajar juntos”: no solo hacer preguntas, debatir y tomar notas, sino hacer preguntas y debatir mientras se escribe el código.

La ley de Conway

En 1967, Melvin Conway introdujo una idea en *How do Committees Invent?* [Con68] que pasó a conocerse como ley de Conway:

> *Las organizaciones que diseñen sistemas se verán abocadas a producir diseños que serán copias de las estructuras de comunicación de dichas organizaciones.*

Es decir, las estructuras sociales y las rutas de comunicación del equipo y la organización se reflejarán en la aplicación, sitio web o producto que se esté desarrollando. Diversos estudios han demostrado un fuerte respaldo a esta idea. Nosotros hemos sido testigos de ello en primera fila en innumerables ocasiones, por ejemplo, en equipos donde nadie habla con los demás, lo que genera sistemas asilados en silos. O equipos que se dividían en dos, lo que tenía como resultado una división entre cliente/servidor o *front end*/*back end*.
También hay estudios que apoyan el principio inverso: podemos estructurar nuestro equipo de manera deliberada para darle el aspecto que queramos que tenga el código. Por ejemplo, se ha demostrado que los equipos distribuidos a nivel geográfico tienden a crear software más modular y distribuido.
Pero lo más importante es que los equipos de desarrollo que incluyen a usuarios producirán un software que reflejará con claridad esa implicación y, si los equipos no se molestan mucho, también se notará.

Programación en pareja

La programación en pareja es una de las prácticas de la Programación Extrema que ha logrado la popularidad fuera de la propia XP. En la programación en pareja, un desarrollador maneja el teclado y el otro no. Ambos trabajan juntos en el problema y pueden turnarse para escribir como sea necesario.

La programación en pareja tiene muchos beneficios. Personas diferentes aportan perspectivas y experiencias diferentes, técnicas y enfoques distintos para la resolución de problemas y niveles diferentes de atención a cualquier problema dado. El desarrollador que teclea debe concentrarse en los detalles de bajo nivel de sintaxis y estilo de escritura del código, mientras que el

otro desarrollador es libre de considerar el alcance y los problemas de alto nivel. Aunque parezca una distinción pequeña, recuerde que los humanos tenemos un ancho de banda cerebral limitado. Juguetear con la escritura de palabras y símbolos esotéricos que el compilador aceptará a regañadientes requiere una parte considerable de nuestra propia capacidad de procesamiento. Tener el cerebro completo de un segundo desarrollador disponible durante la tarea aporta una mayor capacidad mental para trabajar.

La presión de grupo inherente de una segunda persona ayuda en momentos de debilidad y contra las malas costumbres de llamar a las variables `foo` y cosas de ese tipo. Seremos más reacios a tomar un atajo que pueda ser vergonzoso si hay otra persona mirándonos, lo cual también ayuda a producir un software de mayor calidad.

Programación en grupo

Y si dos cabezas piensan mejor que una, ¿qué le parece tener a una docena de personas distintas trabajando en el mismo problema al mismo tiempo mientras una teclea? La programación en grupo, pese a su nombre, no implica una muchedumbre. Es una extensión de la programación en pareja que implica a más de dos desarrolladores. Sus partidarios afirman haber conseguido grandes resultados al utilizar un grupo para resolver problemas difíciles. Los grupos pueden incluir con facilidad a personas que no suelan considerarse parte del equipo de desarrollo, incluyendo usuarios, patrocinadores del proyecto y encargados de las pruebas. De hecho, en nuestro primer proyecto “imposible” juntos, era habitual que uno de nosotros teclease mientras el otro hablaba del problema con nuestro experto en el negocio. Era un grupo pequeño de tres.

Puede pensar en la programación en grupo como una colaboración estrecha mientras se escribe código en vivo.

¿Qué debería hacer?

Si en la actualidad siempre programa solo, quizá pueda probar la programación en pareja. Dele un mínimo de dos semanas, solo unas horas cada vez, ya que al principio le resultará extraño. Para poner en común

ideas nuevas o diagnosticar problemas peliagudos, tal vez pueda probar una sesión de programación en grupo.

Si ya programa en pareja o en grupo, ¿quién está incluido? ¿Solo hay desarrolladores o permite que participen miembros de su equipo ampliado: usuarios, probadores, patrocinadores...?

Y como ocurre con cualquier colaboración, necesita gestionar los aspectos humanos además de los técnicos. Aquí tiene algunos trucos para empezar:

- Construya el código, no su ego. No va de quién es el más listo; todos tenemos nuestros momentos, buenos y malos.
- Empiece con algo pequeño. Un grupo con solo 4 o 5 personas, o solo unas pocas parejas, en sesiones cortas.
- Critique el código, no a la persona. “Vamos a repasar este bloque” suena mucho mejor que “te equivocas”.
- Escuche e intente entender los puntos de vista de los demás. Diferente no significa equivocado.
- Realice retrospectivas frecuentes para intentar mejorar para la próxima vez.

Escribir código en la misma oficina o de forma remota, solo, en pareja o en grupo son maneras efectivas de trabajar juntos para resolver problemas. Si usted y su equipo lo han hecho siempre de una sola manera, quizá quieran experimentar con un estilo diferente, pero no se lance con un enfoque ingenuo: hay reglas, sugerencias y directrices para cada uno de estos estilos de desarrollo. Por ejemplo, con la programación en grupo, la persona que teclea se cambia cada 5-10 minutos. Lea e investigue, tanto en libros de texto como en informes de experiencias y hágase una idea de las ventajas e inconvenientes que puede encontrar. Puede que quiera empezar por escribir el código de un ejercicio simple y no empezar directamente por su código de producción más difícil. Pero, haga lo que haga, le ofrecemos un último consejo:

Truco 82. No se meta a escribir código usted solo.

48 La esencia de la agilidad

Siempre usas esa palabra, y no creo que signifique lo que tú crees.

—Íñigo Montoya, La princesa prometida

"Ágil" es un adjetivo: es la manera en que se hace algo. Puede ser un desarrollador ágil. Puede estar en un equipo que adopte prácticas ágiles, un equipo que responda al cambio y los contratiempos con agilidad. La agilidad es su estilo, no usted.

Truco 83. Ágil no es un nombre; ágil es la manera de hacer las cosas.

Mientras escribimos esto, casi 20 años después del comienzo del Manifiesto por el Desarrollo Ágil de Software,[4] vemos a muchos desarrolladores aplicar con éxito esos valores. Vemos equipos fantásticos que encuentran maneras de tomar esos valores y utilizarlos para que guíen lo que hacen y cómo cambian lo que hacen.

Pero también vemos otra parte de la agilidad. Vemos equipos y empresas que anhelan soluciones listas para usar: Agile-in-a-Box. Y vemos a muchos asesores y empresas demasiado felices de venderles lo que quieren. Vemos empresas que adoptan más capas de dirección, más informes formales, más desarrolladores especializados y más nombres para puestos de trabajo elegantes que solo significan "persona con una tabla sujetapapeles y un cronómetro".[5]

Consideramos que mucha gente ha perdido de vista el verdadero significado de la agilidad, y nos gustaría ver a personas que volviesen a las raíces.

Recuerde los valores del manifiesto:

Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- *Individuos e interacciones sobre procesos y herramientas.*
- *Software funcionando sobre documentación extensiva.*
- *Colaboración con el cliente sobre negociación contractual.*
- *Respuesta ante el cambio sobre seguir un plan.*

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Está claro que cualquiera que le venda algo que aumente la importancia de las cosas de la derecha por encima de las cosas en la izquierda no valora lo mismo que nosotros y los demás creadores del manifiesto.

Y cualquiera que le venda una solución en una caja no ha leído la declaración introductoria. Los valores están motivados y conformados por el acto continuo de descubrir maneras mejores de producir software. No se trata de un documento estático, sino de sugerencias para un proceso generativo.

Nunca puede haber un proceso ágil

De hecho, cada vez que alguien dice “haga esto y será ágil”, se equivoca. Por definición.

Porque la agilidad, tanto en el mundo físico como en el desarrollo de software, tiene que ver con responder al cambio, responder a los elementos desconocidos que encontramos una vez que nos hemos puesto en marcha. Una gacela que va corriendo no va en línea recta. Un gimnasta hace cientos de correcciones en segundos a medida que responde a los cambios en su entorno y los errores mínimos en la colocación de sus pies.

Ocurre lo mismo con equipos y desarrolladores individuales. No hay un único plan que pueda seguir cuando desarrolle software. Tres de los cuatro valores le dicen eso. Tienen que ver con recopilar y responder al *feedback*.

Los valores no le dicen qué hacer. Le dicen qué buscar cuando decida usted mismo qué hacer.

Estas decisiones son siempre contextuales: dependen de quién es usted, la naturaleza de su equipo, su aplicación, sus herramientas, su empresa, su cliente, el mundo exterior; una cantidad increíble de factores, algunos importantes y otros triviales. Ningún plan estático fijo puede sobrevivir a esta incertidumbre.

Entonces, ¿qué hacemos?

Nadie puede decirle qué hacer, pero creemos que podemos decirle algo acerca del espíritu con el que lo hace. Todo se reduce a cómo lidia usted con la incertidumbre. El manifiesto sugiere que esto se hace recopilando

feedback y actuando en consecuencia, así que esta es nuestra receta para trabajar de una manera ágil:

1. Averigüe dónde está.
2. Dar el paso significativo más pequeño hacia donde quiere estar.
3. Evaluar dónde acaba y arreglar cualquier cosa que haya estropeado.

Repita estos pasos hasta que haya acabado y utilícelos de manera recursiva, en cada nivel de todo lo que haga.

A veces, incluso la decisión que parece más trivial se vuelve importante cuando se recopila *feedback*.

"Ahora mi código necesita obtener el propietario de la cuenta.

```
let user = accountOwner(accountID);
```

Hmmm... 'user' es un nombre inútil. Voy a cambiarlo por 'owner'.

```
let owner = accountOwner(accountID);
```

Pero ahora resulta un poco redundante. ¿Qué estoy intentando hacer en realidad? La historia dice que estoy enviando a esta persona un correo electrónico, así que necesito encontrar su dirección. Quizá no necesite al propietario de la cuenta completo.

```
let email = emailOfAccountOwner(accountID);
```

Al aplicar el bucle de *feedback* a un nivel realmente bajo (el poner el nombre a una variable), en realidad hemos mejorado el diseño del sistema global, reduciendo el acoplamiento entre este código y el código que se ocupa de las cuentas.

El bucle de *feedback* también se aplica al nivel más alto de un proyecto. Hemos hecho algunos de nuestros trabajos de mayor éxito cuando hemos empezado a trabajar en los requisitos de un cliente, hemos dado un solo paso y nos hemos dado cuenta de que lo que estábamos a punto de hacer no era necesario, que la mejor solución ni siquiera implicaba software.

Este bucle se aplica fuera del ámbito de un solo proyecto. Los equipos deberían aplicarlo para revisar su proceso y ver qué tal ha funcionado. Un equipo que no experimenta de manera continua con su proceso no es un equipo ágil.

Y esto dirige el diseño

En el tema 8, “La esencia del buen diseño”, afirmábamos que la medida del diseño es lo fácil que es cambiar el resultado de ese diseño: un buen diseño produce algo que es más fácil de cambiar que un mal diseño.

Y esta sección sobre la agilidad explica por qué es así.

Hace un cambio y descubre que no le gusta. En el paso 3 de la lista dije que debemos ser capaces de arreglar lo que estropeemos. Para hacer que nuestro bucle de *feedback* sea eficiente, esta reparación debe ser lo menos dolorosa posible. Si no es así, sentiremos la tentación de no hacerle caso y dejar el problema sin arreglar. Hablamos de esto en el tema 3, “Entropía del software”. Para hacer que esto de la agilidad funcione, necesitamos practicar el buen diseño, porque este hace que las cosas sean fáciles de cambiar. Y, si algo es fácil de cambiar, podemos realizar ajustes en cualquier nivel, sin dudar.

Eso es la agilidad.

Las secciones relacionadas incluyen

- Tema 27, “No vaya más rápido que sus faros”.
- Tema 40, “Refactorización”.
- Tema 50, “Los cocos no sirven”.

Retos

El bucle de *feedback* simple no sirve solo para el software. Piense en otras decisiones que haya tomado hace poco. ¿Podría haberse mejorado alguna si hubiese pensado en cómo podría deshacer lo que ha hecho si las cosas no le llevasen en la dirección en la que iba? ¿Se le ocurren maneras de poder mejorar lo que hace recopilando *feedback* y actuando en consecuencia?

---

[1] ¿Le parece que una semana es mucho tiempo? En realidad, no lo es, sobre todo si está fijándose en procesos en los que la dirección y los trabajadores ocupan mundos diferentes. La dirección le

ofrecerá una visión de cómo funcionan las cosas, pero, cuando baje a la planta, se encontrará con una realidad muy diferente, una que le llevará un tiempo asimilar.

[2] https://www.wired.com/1999/01/eno/.

[3] https://www.psychologytoday.com/us/blog/your-brain-work/201209/stop-trying-solve-problems.

[4] https://agilemanifesto.org/iso/es/manifesto.html.

[5] Para leer más acerca de lo malo que puede ser este enfoque, consulte *The Tyranny of Metrics* [Mul18].

9
Proyectos pragmáticos

Cuando el proyecto se pone en marcha, necesitamos alejarnos de las cuestiones de filosofía y creación de código individuales y para hablar de asuntos más amplios, del tamaño del proyecto. No vamos a entrar en cuestiones específicas de la gestión de proyectos, pero hablaremos de varias áreas cruciales que pueden hacer que cualquier proyecto triunfe o se hunda.

En cuanto tenga más de una persona trabajando en un proyecto, tendrá que establecer algunas reglas básicas y delegar partes del proyecto en consecuencia. En “Equipos pragmáticos”, le mostraremos cómo hacerlo mientras respeta la filosofía pragmática.

El propósito de un método de desarrollo de software es ayudar a las personas a trabajar juntas. ¿Están usted y su equipo haciendo lo que les funciona bien o solo están invirtiendo en los artefactos superficiales triviales sin obtener los beneficios reales que merecen? Veremos por qué “Los cocos no sirven” y ofreceremos el verdadero secreto del éxito.

Y, por supuesto, nada de eso importa si no puede entregar software de manera consistente y fiable. Esa es la base del trío mágico del control de versiones, las pruebas y la automatización: el “Kit pragmático básico”.

Sin embargo, al final, el éxito está en los ojos del que mira: el patrocinador del proyecto. La percepción del éxito es lo que cuenta, y en “Deleite a sus usuarios” le mostraremos cómo hacer que todos los patrocinadores del proyecto estén satisfechos.

El último truco del libro es una consecuencia directa de todo lo demás. En “Orgullo y prejuicio”, le pedimos que firme su trabajo y que se enorgullezca de lo que hace.

49 Equipos pragmáticos

En Group L, Stoffel supervisa a seis programadores de primera categoría, un reto de gestión casi comparable a pastorear gatos.

—The Washington Post Magazine, 9 de junio, 1985

Incluso en 1985, el chiste sobre pastorear gatos ya estaba quedándose obsoleto. Para cuando publicamos la primera edición con el cambio de siglo, era muy viejo. Y, sin embargo, resiste, porque hay algo de cierto en él. Los programadores se parecen un poco a los gatos: inteligentes, tenaces, obstinados, independientes y, con frecuencia, adorados en la red.

Hasta ahora, en este libro hemos visto técnicas pragmáticas que ayudan a un individuo a ser mejor programador. ¿Pueden funcionar estos métodos también para equipos, incluso para equipos de personas tercas e independientes? La respuesta es un "¡sí!" rotundo. Hay ventajas en ser un individuo pragmático, pero estas ventajas se multiplican por mucho si el individuo trabaja en un equipo pragmático.

Desde nuestro punto de vista, un equipo es una entidad propia, pequeña y mayormente estable. Cincuenta personas no son un equipo, son una horda.[1] Los equipos en los que a los miembros se les arrastra a otras tareas todo el tiempo y nadie conoce a los demás tampoco son un equipo, son solo unos extraños que comparten de forma temporal la parada del autobús bajo la lluvia.

Un equipo pragmático es pequeño, con menos de 10-12 personas. Los miembros apenas van y vienen. Todos conocen bien a todos, confían en los demás y dependen unos de otros.

Truco 84. Mantenga equipos pequeños y estables.

En esta sección, veremos con brevedad cómo pueden aplicarse las técnicas pragmáticas a equipos como conjuntos. Estas notas son solo un comienzo. Una vez que tengamos a un grupo de desarrolladores pragmáticos trabajando en un entorno propicio, ellos desarrollarán y perfeccionarán enseguida sus propias dinámicas de equipo que les funcionen.

Vamos a retomar algunas de las secciones anteriores aplicadas ahora a equipos.

Sin ventanas rotas

La calidad es una cuestión de equipo. Incluso el desarrollador más diligente puesto en un equipo al que le da igual encontrará dificultades para mantener el entusiasmo necesario para solucionar problemas triviales. El problema se agrava si el equipo disuade de forma activa al desarrollador de dedicar tiempo a esos arreglos.

Los equipos como conjunto no deberían tolerar ventanas rotas, esas pequeñas imperfecciones que nadie repara. El equipo debe asumir la responsabilidad de la calidad del producto, apoyando a los desarrolladores que entienden la filosofía de que no haya ventanas rotas que describimos en el tema 3, “Entropía del software”, y animando a aquellos que aún no la han descubierto.

Algunas metodologías de equipo tienen un “responsable de la calidad”, alguien en quien el equipo delega la responsabilidad de la calidad del producto final. Está claro que esto es ridículo: la calidad solo puede venir de las contribuciones individuales de todos los miembros del equipo. La calidad está integrada, no acoplada después como un elemento extra.

Ranas hervidas

¿Recuerda a la rana apócrifa en la cazuela con agua del tema 4, “Sopa de piedras y ranas hervidas”? No nota el cambio gradual en su entorno y acaba cocida. Lo mismo puede ocurrir a los individuos que no están atentos. Puede ser difícil prestar atención al entorno global cuando estamos inmersos de pleno en el desarrollo de un proyecto.

Es aún más fácil para un equipo en conjunto acabar hervido. Una persona asume que otra está ocupándose de un problema o que el líder del equipo debe haber aprobado un cambio que el usuario está solicitando. Incluso los equipos con las mejores intenciones pueden no estar al tanto de cambios significativos en sus proyectos.

Luche contra esto. Anime a todo el mundo a monitorizar de manera activa el entorno para ver si hay cambios, características adicionales, entornos. Manténgase alerta para detectar posibles ampliaciones del alcance, reducciones en las escalas de tiempo características adicionales, entornos nuevos... Cualquier cosa que no estuviese dentro de su

comprensión original. Tenga métricas sobre los nuevos requisitos.[2] El equipo no tiene por qué rechazar los cambios sin más, solo tiene que ser consciente de que están produciéndose. De lo contrario, estará en agua caliente.

Planifique su cartera de conocimientos

En el tema 6, "Su cartera de conocimientos", veíamos maneras de las que deberían invertir en su cartera de conocimientos personal a su propio ritmo. Los equipos que deseen el éxito también necesitan tener en cuenta su inversión en conocimientos y habilidades.

Si su equipo se toma en serio la mejora y la innovación, necesita planificarlo. Intentar hacer cosas "cuando haya un ratito libre" significa que nunca se harán. Da igual el tipo de trabajos pendientes, listas de tareas o flujo con el que esté trabajando, no lo reserve solo para el desarrollo de funcionalidades. El equipo no trabaja solo en características nuevas. Algunos ejemplos posibles son:

- Mantenimiento de sistemas antiguos: Aunque nos encante trabajar en sistemas nuevos y relucientes, es probable que haya que realizar tareas de mantenimiento en nuestro sistema antiguo. Hemos conocido a equipos que intentan arrinconar este trabajo. Si se encarga al equipo que se ocupe de estas tareas, que lo haga, en serio.
- Reflexión sobre el proceso y perfeccionamiento: La mejora continua solo puede producirse cuando dedicamos un tiempo a fijarnos en lo que hay a nuestro alrededor para ver lo que funciona y lo que no y después hacemos cambios (consulte el tema 48, "La esencia de la agilidad"). Hay demasiados equipos que están tan ocupados achicando el agua que no tienen tiempo para reparar la fuga. Prográmelo. Arréglela.
- Experimentos con nuevas tecnologías: No adopte nuevas tecnologías, *frameworks* o bibliotecas solo porque "lo está haciendo todo el mundo" o basándose en algo que vio en una conferencia o leyó por Internet. Investigue de forma deliberada las tecnologías candidatas

con prototipos. Apunte en la agenda tareas para probar cosas nuevas y analizar resultados.

- Aprendizaje y mejora de las habilidades: El aprendizaje y las mejoras personales son un comienzo estupendo, pero muchas habilidades son a menudo más efectivas cuando se extienden por todo el equipo. Planifique hacerlo, ya sea mediante almuerzos informativos informales o a través de sesiones de formación más formales.

Truco 85. Planifíquelo para hacer que ocurra.

Comunique la presencia del equipo

Es obvio que los desarrolladores de un equipo deben hablar entre sí. Ofrecemos algunas sugerencias para facilitar esto en el tema 7, "¡Comuníquese!". Sin embargo, es fácil olvidar que el equipo en sí tiene una presencia dentro de la organización. El equipo como entidad necesita comunicarse de manera clara con el resto del mundo.

Para las personas de fuera, los peores equipos de proyecto son aquellos que parecen huraños y reservados. Celebran reuniones sin estructura, donde nadie quiere hablar. Sus correos electrónicos y los documentos del proyecto son un caos: no hay dos iguales y cada uno utiliza una terminología diferente.

Los equipos de proyecto estupendos tienen una personalidad distinta. La gente está desando tener reuniones con ellos, porque sabe que verá una presentación bien preparada que hará que todo el mundo se sienta bien. La documentación que producen está bien definida, es exacta y coherente. El equipo habla con una sola voz.[3] Puede que incluso tenga sentido del humor.

Hay un sencillo truco de marketing que ayuda a los equipos a comunicar como uno solo: generar una marca. Cuando empiece un proyecto, piense un nombre, a ser posible algo excéntrico. (En el pasado, hemos puesto a proyectos nombres como loros asesinos que atacan a ovejas, ilusiones ópticas, jerbos, personajes de dibujos animados y ciudades míticas). Dedique 30 minutos a pensar en un logo alocado y utilícelo. Use el nombre de su equipo sin reparos cuando hable con la gente. Parece una tontería,

pero eso da a su equipo una identidad sobre la que construir y al mundo algo memorable para asociar con su trabajo.

No se repitan

En el tema 9, “DRY: los males de la duplicación”, hemos hablado acerca de las dificultades de eliminar trabajo duplicado entre miembros de un equipo. Esta duplicación lleva a un esfuerzo desperdiciado y puede generar una pesadilla de mantenimiento. Los sistemas aislados o en silos son comunes en estos equipos, donde se comparte poca información y hay mucha funcionalidad duplicada. La buena comunicación es clave para evitar estos problemas. Y por “buena” queremos decir instantánea y sin fisuras.

Debería poder hacer una pregunta a los miembros del equipo y recibir una respuesta más o menos instantánea. Si el equipo se encuentra en la misma ubicación, puede ser algo tan simple como asomar la cabeza en el cubículo de al lado o recorrer el pasillo. Para los equipos remotos, puede que haya que depender de aplicaciones de mensajería u otros medios electrónicos.

Si tiene que esperar una semana a que el equipo se reúna para hacer una pregunta o compartir un estado, eso es una fisura enorme.[4] “Sin fisuras” significa que hacer preguntas, compartir el progreso, los problemas, las opiniones y el aprendizaje y estar al tanto de lo que hacen los compañeros del equipo es fácil y no requiere ceremonias.

Preste atención para respetar el principio DRY.

Balas trazadoras en equipos

Un equipo de proyecto tiene que realizar muchas tareas diferentes en distintas áreas del proyecto, tocando un montón de tecnologías diferentes. Entender los requisitos, diseñar la arquitectura, escribir código para el *front end* y el servidor, hacer las pruebas, todo tiene que ocurrir. Pero es una confusión habitual pensar que estas actividades y tareas pueden producirse por separado, de manera aislada. No pueden.

Algunas metodologías abogan por utilizar todo tipo de roles y títulos diferentes dentro del equipo o por crear equipos especializados totalmente independientes, pero el problema de ese enfoque es que introduce puertas y traspasos. Ahora, en vez de un flujo regular desde el equipo al despliegue, tiene puertas artificiales donde se detiene el trabajo. Hay que esperar a que se acepten los traspasos. Aprobaciones. Papeleo. Los partidarios del desarrollo Lean llaman a esto desperdicio y se esfuerzan por eliminarlo de manera activa.

Todas estas funciones y actividades son en realidad perspectivas diferentes del mismo problema, y separarlas de forma artificial puede generar muchísimos problemas. Por ejemplo, es poco probable que los programadores que están dos o tres niveles alejados de los usuarios reales de su código estén al tanto del contexto en el que se utiliza su trabajo. No serán capaces de tomar decisiones bien fundamentadas.

En “Balas trazadoras”, recomendamos desarrollar funcionalidades individuales, por muy pequeñas y limitadas que sean en principio, que vayan de extremo a extremo por todo el sistema. Eso significa que necesitamos todas las habilidades que hacen eso dentro del equipo: *front end*, UI/UX, servidor, administrador de base de datos, QA, etc., todos ellos cómodos y acostumbrados a trabajar juntos. Con el enfoque de las balas trazadoras, puede implementar porciones de funcionalidad muy pequeñas y recibir *feedback* inmediato acerca de qué tal se comunica su equipo y cumple sus funciones. Eso crea un entorno en el que puede hacer cambios y ajustar el equipo y el proceso con rapidez y facilidad.

Truco 86. Organice equipos completamente funcionales.

Construya los equipos de manera que pueda crear el código de extremo a extremo, de manera gradual e iterativa.

Automatización

Una manera estupenda de garantizar la coherencia y la exactitud es automatizar todo lo que hace el equipo. ¿Por qué pelearse con los estándares de formato del código cuando su editor o IDE puede hacerlo por

usted de forma automática? ¿Por qué realizar pruebas manuales cuando la construcción continua puede ejecutar pruebas automáticamente? ¿Por qué desplegar a mano cuando la automatización puede hacerse de la misma manera cada vez, de forma repetible y fiable?

La automatización es un componente esencial de cualquier equipo de proyecto. Asegúrese de que el equipo tiene habilidades de creación de herramientas para construir y desplegar las herramientas que automatizan el desarrollo del proyecto y el despliegue a producción.

Saber cuándo dejar de añadir pintura

Recuerde que los equipos están formados por individuos. Dé a cada miembro la capacidad para brillar por su cuenta. Deles la estructura suficiente para apoyarlos y para garantizar que el proyecto entrega valor. Después, como el pintor de "Software lo bastante bueno", resista la tentación de añadir más pintura.

Las secciones relacionadas incluyen

- Tema 2, "El gato se comió mi código fuente".
- Tema 7, "¡Comuníquese!".
- Tema 12, "Balas trazadoras".
- Tema 19, "Control de versiones".
- Tema 50, "Los cocos no sirven".
- Tema 51, "Kit pragmático básico".

Retos

- Fíjese en equipos de éxito fuera del área del desarrollo de software. ¿Qué hace que tengan éxito? ¿Utilizan alguno de los procesos descritos en esta sección?
- La próxima vez que empiece un proyecto, intente convencer a la gente de que lo conviertan en una marca. Dé a la organización tiempo para acostumbrarse a la idea y, después, haga una auditoría rápida

para ver qué diferencia ha marcado, tanto dentro del equipo como a nivel externo.

- Es probable que alguna vez le hayan planteado problemas como “Si 4 obreros necesitan 6 horas para cavar una zanja, ¿cuánto tardarían 8 obreros?”. Sin embargo, en la vida real, ¿qué factores afectan a la respuesta si en vez de eso los trabajadores estuviesen escribiendo código? ¿En cuántos escenarios se reduce de verdad el tiempo?
- Lea *The Mythical Man Month* [Bro96] de Frederick Brooks. Para obtener un punto extra, compre dos copias para poder leerlo el doble de rápido.

50 Los cocos no sirven

Los nativos isleños nunca habían visto un avión antes ni habían conocido a nadie como aquellos forasteros. A cambio del uso de su tierra, los forasteros proporcionaban pájaros mecánicos que llegaban y se iban volando todo el día en una “pista” y traían una riqueza material increíble a su isla. Los forasteros mencionaron algo sobre una guerra y sobre luchar. Un día, se acabó y se marcharon, llevándose con ellos sus extrañas riquezas.

Los isleños estaban desesperados por recuperar su buena fortuna y reconstruyeron una copia del aeropuerto, la torre de control y los equipos usando materiales locales: enredaderas, cáscaras de coco, hojas de palmera y cosas así. Pero, por alguna razón, incluso aunque tenían todo en su sitio, los aviones no llegaron. Habían imitado la forma, pero no el contenido. Los antropólogos denominaron a esto un “culto cargo”.

Con demasiada frecuencia, somos los isleños.

Resulta fácil y tentador caer en la trampa del culto cargo: al invertir en los artefactos que se ven con facilidad y desarrollarlos, esperamos atraer a la magia subyacente en funcionamiento, pero, como ocurre con los cultos cargo originales de Melanesia,[5] un aeropuerto falso lleno de cáscaras de coco no puede sustituir a uno real.

Por ejemplo, hemos visto personalmente equipos que afirmaban estar utilizando Scrum. Pero, al examinar su trabajo más de cerca, resultaba que estaban celebrando una reunión de pie diaria una vez a la semana, con

iteraciones de cuatro semanas que a menudo se convertían en iteraciones de seis u ocho semanas. Les parecía que eso estaba bien porque estaba utilizando una herramienta de planificación "ágil" popular. Solo estaban invirtiendo en los artefactos superficiales e incluso entonces solo en el nombre, como si "de pie" o "iteración" fuesen algún tipo de mantra para los supersticiosos. No era de extrañar que ellos tampoco lograsen atraer a la magia real.

El contexto importa

¿Alguna vez ha caído usted o su equipo en esta trampa? Pregúntese por qué está utilizando siquiera ese método de desarrollo en particular. O ese *framework*. O esa técnica de pruebas. ¿Es adecuado eso en realidad para el trabajo que tiene entre manos? ¿Funciona bien para usted o se ha adoptado solo porque es lo que estaba utilizándose en la última historia de éxito extendida por Internet? Existe una tendencia actual de adoptar políticas y procesos de empresas de éxito como Spotify, Netflix, Stripe, GitLab y otras. Cada una tiene su punto de vista único sobre el desarrollo de software y la gestión. Pero piense en el contexto: ¿está usted en el mismo mercado, con las mismas restricciones y oportunidades, un tamaño de organización y un nivel de experiencia similares, una gestión parecida y una cultura similar? ¿Una base de usuarios y unos requisitos parecidos?

No se deje engañar. Los artefactos particulares, los métodos, los procesos, las políticas y las estructuras superficiales no bastan.

Truco 87. Haga lo que funcione, no lo que esté de moda.

¿Cómo sabe "lo que funciona"? Gracias a la más fundamental de las técnicas pragmáticas:

Pruébelo.

Haga una prueba piloto de la idea con un equipo o conjunto de equipos pequeño. Mantenga las partes buenas que parezcan funcionar bien y descarte todo lo demás como desperdicio o tara. Nadie degradará su organización porque no opere de la misma manera que Spotify o Netflix, porque ni siquiera ellas seguían sus procesos actuales cuando estaban

creciendo. Y dentro de unos años, cuando esas empresas maduren, pivoten y sigan desarrollándose, estarán haciendo algo diferente otra vez.

Ese es el secreto real de su éxito.

La talla única no le queda bien a nadie

La finalidad de una metodología de desarrollo de software es ayudar a las personas a trabajar juntas. Como explicamos en “La esencia de la agilidad”, no hay un solo plan que pueda seguirse cuando se desarrolla software, sobre todo, si es un plan que se le ha ocurrido a alguien en otra empresa.

Muchos programas de certificación son aún peor que eso: se basan en que el alumno sea capaz de memorizar y seguir las reglas. Pero eso no es lo que le conviene. Necesita la capacidad de ver más allá de las reglas existentes y explotar las posibilidades para conseguir ventaja. Es una mentalidad muy diferente a “pero Scrum/Lean/Kanban/XP/el desarrollo ágil lo hace así...”, etc.

En vez de eso, le conviene tomar los mejores elementos de una metodología en particular y adaptarlos para su uso. No hay una talla única, y los métodos actuales están muy lejos de ser completos, así que tendrá que fijarse en más de un método popular.

Por ejemplo, Scrum define algunas prácticas de gestión de proyectos, pero Scrum en sí mismo no proporciona orientación suficiente a nivel técnico para equipos o a nivel de cartera/gobernanza para la dirección. Entonces, ¿por dónde empezamos?

¡Sea como ellos!

Con frecuencia, oímos a los jefes de desarrollo de software decir a sus empleados: “Deberíamos operar como Netflix” (u otra de esas empresas líderes). Por supuesto, podría hacerlo.
Primero, hágase con varios cientos de miles de servidores y decenas de millones de usuarios...

El objetivo real

Por supuesto, el objetivo no es “hacer Scrum”, “hacer desarrollo ágil”, “hacer Lean” o lo que sea. El objetivo es estar en una posición que permita entregar software funcional que dé a los usuarios alguna nueva capacidad en cualquier momento. No dentro de unas semanas, unos meses o unos años, sino ahora. Para muchos equipos y organizaciones, la entrega continua parece una meta elevada e inalcanzable, sobre todo si les endosan un proceso que restringe la entrega a meses o semanas. Pero, como ocurre con cualquier objetivo, la clave es seguir apuntando en la dirección adecuada (véase la figura 9.1).

Si entrega en años, intente acortar el ciclo a meses. De meses, acórtelo a semanas. De un *sprint* de cuatro semanas, pruebe a reducir a dos. De un *sprint* de dos semanas, pruebe a pasar a una. Después, a diario y, por último, cuando se le pida. Tenga en cuenta que ser capaz de entregar bajo demanda no significa que esté obligado a entregar cada minuto del día. Entregue cuando los usuarios lo necesiten, cuando hacerlo tiene sentido para el negocio.

Truco 88. Entregue cuando los usuarios lo necesiten.

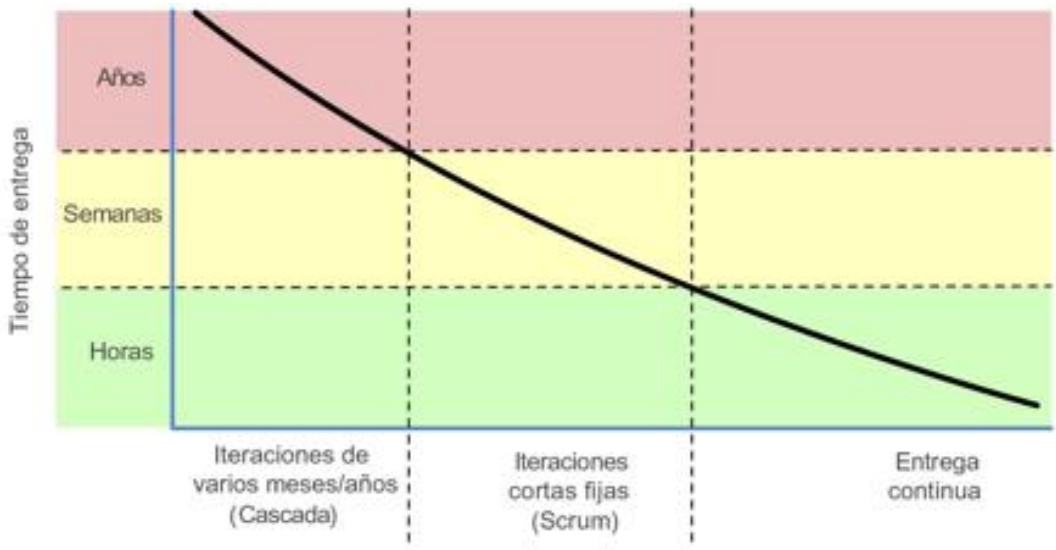


Figura 9.1.

Para poder pasar a este estilo de desarrollo continuo, necesita una infraestructura sólida como una roca, de la que hablaremos en el siguiente

tema, “Kit pragmático básico”. El desarrollo se hace en el tronco principal del sistema de control de versiones, no en ramas, y se utilizan técnicas como los *feature switches* para pasar características de prueba a los usuarios de manera selectiva.

Una vez que tenga la infraestructura en orden, necesita decidir cómo organizar el trabajo. Puede que los principiantes quieran empezar con Scrum para la gestión de proyectos, más las prácticas técnicas de la Programación Extrema (XP). Los equipos más disciplinados y experimentados pueden preferir técnicas Lean y Kanban, tanto para el equipo como para, tal vez, cuestiones de gobernanza más amplias.

Pero no se lo crea solo porque lo decimos nosotros. Investigue y pruebe estos enfoques usted mismo, pero tenga cuidado de no pasarse. Invertir en exceso en una metodología particular puede hacer que pase por alto las alternativas. Se acostumbrará a ella. Pronto le resultará difícil ver cualquier otra opción. Se anquilosará y ya no podrá adaptarse con rapidez. Para eso, le daría igual utilizar cocos.

Las secciones relacionadas incluyen

- Tema 12, “Balas trazadoras”.
- Tema 27, “No vaya más rápido que sus faros”.
- Tema 48, “La esencia de la agilidad”.
- Tema 49, “Equipos pragmáticos”.
- Tema 51, “Kit pragmático básico”.

51 Kit pragmático básico

> *La civilización avanza ampliando el número de operaciones importantes que podemos realizar sin pensar.*
>
> *—Alfred North Whitehead*

Cuando los coches eran una novedad, las instrucciones para arrancar un Ford Modelo T ocupaban más de dos páginas. Con los coches modernos, solo hay que pulsar un botón; el procedimiento de arranque es automático y

a prueba de tontos. Una persona siguiendo una lista de instrucciones podría inundar el motor, pero el arranque automático no lo hará.

Aunque el desarrollo de software es todavía una industria en la fase del Modelo T, no podemos permitirnos leer dos páginas de instrucciones una y otra vez para una operación común. Ya sea el procedimiento de construcción y liberación, las pruebas, el papeleo del proyecto o cualquier otra tarea recurrente del proyecto, tiene que ser automático y repetible en cualquier máquina capaz. Además, queremos garantizar la consistencia y la repetibilidad en el proyecto. Los procedimientos manuales dejan la consistencia al azar; la repetibilidad no está garantizada, sobre todo si hay aspectos del procedimiento abiertos a la interpretación que hagan personas diferentes.

Después de escribir la primera edición de *El programador pragmático*, queríamos crear más libros para ayudar a los equipos a desarrollar software. Supusimos que deberíamos empezar por el principio: cuáles son los elementos más básicos e importantes que todo equipo necesita, al margen de la metodología, el lenguaje o *stack* tecnológico. Así nació la idea del kit pragmático básico, cubriendo estos tres temas cruciales e interrelacionados:

- Control de versiones.
- Pruebas de regresión.
- Automatización completa.

Estos son los tres pilares que sostienen todo proyecto. Vamos a ver cómo.

Dirija con el control de versiones

Como decíamos en “Control de versiones”, le conviene mantener todo lo necesario para construir el proyecto en un sistema de control de versiones. Esa idea se vuelve aún más importante en el contexto del proyecto en sí.

Primero, permite crear máquinas que sean efímeras. En vez de una máquina sagrada y destartalada en la esquina de la oficina que nadie se atreve a tocar,[6] las máquinas o clústeres de construcción se crean bajo demanda como instancias de *spot* en la nube. La configuración del

despliegue también está en el control de versiones, así que el paso a producción puede gestionarse de manera automática.

Y esa es la parte importante: a nivel del proyecto, el control de versiones dirige el proceso de construcción y liberación.

Truco 89. Use el control de versiones para dirigir construcciones, pruebas y liberaciones.

Es decir, la construcción, las pruebas y el despliegue se desencadenan mediante *commits* o *pushes* al sistema de control de versiones y se integran en un contenedor en la nube. La liberación a preproducción o a producción se especifica utilizando una etiqueta en el sistema de control de versiones. Así, las liberaciones se convierten en parte del día a día con mucha menos ceremonia; es una entrega continua, que no está atada a ninguna máquina de construcción ni a la máquina de ningún desarrollador.

Pruebas despiadadas y continuas

Muchos desarrolladores hacen pruebas con delicadeza, sabiendo de manera subconsciente dónde va a romperse el código y evitando los puntos débiles. Los programadores pragmáticos somos diferentes. Estamos decididos a encontrar fallos ahora para no tener que soportar la vergüenza de que otros los encuentren después.

Encontrar fallos es parecido a pescar con una red. Utilizamos redes pequeñas y finas (pruebas unitarias) para atrapar a los pececillos y redes grandes y toscas para atrapar a los tiburones enormes. A veces, el pez logra escapar, así que arreglamos los agujeros que encontramos, con la esperanza de atrapar cada vez más defectos escurridizos que van nadando por el mar de nuestro proyecto.

Truco 90. Pruebe pronto, pruebe a menudo, pruebe de forma automática.

Queremos empezar las pruebas en cuanto tenemos el código. Esos pececillos diminutos tienen la mala costumbre de convertirse en tiburones

gigantes devoradores de hombres con rapidez y pescar un tiburón es bastante más difícil. Así pues, escribimos pruebas unitarias. Un montón de pruebas unitarias.

De hecho, un buen proyecto puede tener más código de pruebas que código de producción. El tiempo que se necesita para producir este código de pruebas merece la pena. Acaba siendo mucho más barato a largo plazo y tenemos de verdad la oportunidad de producir un producto con casi cero defectos.

Además, saber que hemos pasado la prueba nos da un alto grado de confianza en que una porción de código está “acabada”.

Truco 91. El código no está acabado hasta que no se ejecutan todas las pruebas.

La construcción automática ejecuta todas las pruebas disponibles. Es importante tener como objetivo “probar de verdad” o, dicho de otro modo, el entorno de pruebas debería ser lo más parecido posible al entorno de producción. Los agujeros son caldo de cultivo para los fallos.

La construcción puede cubrir varios tipos principales de pruebas de software: pruebas unitarias, pruebas de integración, validación y verificación y pruebas de rendimiento. Esta lista no está completa, ni mucho menos, y algunos proyectos especializados pueden requerir varios tipos de pruebas más, pero es un buen punto de partida.

Pruebas unitarias

Una prueba unitaria es código que ejercita un módulo. Hemos hablado de esto en el tema 41, “Probar para escribir código”. Las pruebas unitarias son los cimientos de todas las demás formas de pruebas que describiremos en esta sección. Si las partes no funcionan por sí solas, es probable que no funcionen bien juntas. Todos los módulos que esté utilizando deben pasar sus propias pruebas unitarias antes de poder continuar.

Una vez que todos los módulos pertinentes han pasado sus pruebas individuales, estamos listos para la siguiente fase. Hay que probar el modo en que todos los módulos utilizan e interactúan con los demás a través del sistema.

Pruebas de integración

Las pruebas de integración muestran que los subsistemas principales que conforman el proyecto funcionan e interactúan bien entre sí. Con contratos buenos en su sitio y bien probados, cualquier problema de integración puede detectarse con facilidad. De lo contrario, la integración se convierte en un terreno fértil para los fallos. De hecho, es a menudo la fuente más grande de los fallos del sistema. En realidad, las pruebas de integración son solo una extensión de las pruebas unitarias que hemos descrito; solo estamos probando cómo subsistemas completos respetan sus contratos.

Validación y verificación

En cuanto tenemos un prototipo o una interfaz de usuario ejecutable, debemos responder una pregunta muy importante: los usuarios nos dijeron lo que querían, pero ¿es eso lo que necesitan?

¿Cumple los requisitos funcionales del sistema? Eso es algo que también debe probarse. Un sistema libre de fallos que responde a la pregunta equivocada no es muy útil. Tenemos que ser conscientes de los patrones de acceso del usuario final y de cómo difieren de los datos de prueba del desarrollador (para ver un ejemplo, consulte la historia de las pinceladas del tema 20, “Depuración”).

Pruebas de rendimiento

Las pruebas de rendimiento o de estrés pueden ser también aspectos importantes del proyecto.

Pregúntese si el software cumple los requisitos de rendimiento en condiciones del mundo real, con el número esperado de usuario, conexiones o transacciones por segundo. ¿Es escalable?

Para algunas aplicaciones, puede que necesite un hardware o software de pruebas especializado para simular la carga de manera realista.

Probar las pruebas

Como no podemos escribir software perfecto, tampoco podemos escribir software de pruebas perfecto. Necesitamos probar las pruebas.

Piense en nuestro conjunto de *suites* de pruebas como en un sistema de seguridad elaborado, diseñado para que suene la alarma cuando aparezca un fallo. ¿Qué mejor forma de probar que un sistema de seguridad funciona que intentar entrar por la fuerza?

Una vez que haya escrito una prueba para detectar un fallo concreto, provoque ese fallo de manera intencionada y asegúrese de que la prueba lo detecta. Esto garantiza que la prueba encontrará el fallo si se produce de verdad.

Truco 92. Use saboteadores para probar sus pruebas.

Si va en serio de verdad con las pruebas, tome una rama separada del árbol fuente, introduzca fallos a propósito y verifique que las pruebas los encuentran. A un nivel superior, puede utilizar algo como Chaos Monkey de Netflix[7] para interrumpir (es decir, "matar") servicios y probar la resiliencia de su aplicación.

Cuando escriba pruebas, asegúrese de que las alarmas suenan cuando deberían.

Pruebe a conciencia

Una vez que tenga la confianza de que las pruebas son correctas y están encontrando los fallos que crea, ¿cómo sabe si ha probado la base de código con la minuciosidad suficiente? La respuesta corta es "no lo sabe", y nunca lo sabrá. Podría probar a utilizar herramientas de análisis de la cobertura que observan el código durante las pruebas y hacen un seguimiento de qué líneas de código se han ejecutado y cuáles no. Estas herramientas ayudan a tener una impresión general de lo completas que son las pruebas, pero no espere ver una cobertura del 100 %.[8]

Incluso aunque acierte en cada línea de código, eso no es todo el conjunto. Lo que importa es el número de estados que puede tener su programa. Los estados no son equivalentes a líneas de código. Por ejemplo,

supongamos que tiene una función que toma dos enteros, cada uno de los cuales puede ser un número entre 0 y 999:

```
int test(int a, int b) {
return a / (a + b);
}
```

En teoría, esta función de tres líneas tiene 1.000.000 millón de estados lógicos, de los cuales 999.999 funcionarán de manera correcta y uno no lo hará (cuando `a + b` sea igual a cero). Simplemente saber que ha ejecutado esa línea de código no le dice eso; necesitaría identificar todos los estados posibles del programa. Por desgracia, en general, este es un problema muy difícil. Difícil del tipo "El sol se convertirá en una masa dura y helada antes de que pueda resolverlo".

Truco 93. Pruebe la cobertura de los estados, no la cobertura del código.

Pruebas basadas en propiedades

Una manera estupenda de explorar cómo maneja el código los estados inesperados es hacer que un ordenador genere esos estados.

Utilice técnicas de pruebas basadas en propiedades para generar datos de prueba de acuerdo con los contratos y las invariantes del código que se prueba. Hablamos de esta cuestión en detalle en el tema 42, "Pruebas basadas en propiedades"

Reforzar la red

Por último, nos gustaría revelar el concepto más importante de las pruebas. Es uno muy obvio y prácticamente todos los libros de texto dicen que hay que hacerlo así, pero, por alguna razón, la mayoría de los proyectos siguen sin hacerlo.

Si un fallo se cuela a través de la red de pruebas existentes, hay que añadir una prueba nueva para que lo atrape la próxima vez.

Truco 94. Encuentre fallos una vez.

Una vez que un probador humano encuentra un fallo, debería ser la última vez que un probador humano encuentra ese fallo. Las pruebas automatizadas deberían modificarse para buscar ese fallo concreto a partir de ese momento, cada vez, sin excepciones, no importa lo trivial que sea ni cuánto se queje el desarrollador y diga: “Oh, eso no volverá a ocurrir nunca”.

Porque volverá a ocurrir, y no tenemos tiempo de estar persiguiendo fallos que las pruebas automatizadas podrían haber encontrado por nosotros. Tenemos que dedicar nuestro tiempo a escribir código nuevo (y nuevos fallos).

Automatización completa

Como hemos dicho al principio de esta sección, el desarrollo moderno depende de procedimientos automáticos planeados. Tanto si utiliza algo tan simple como *scripts* del intérprete de comandos con rsync y ssh, como si usa soluciones integrales como Ansible, Puppet, Chef o Salt, no dependa de intervenciones manuales.

Hace mucho tiempo, estábamos en las instalaciones de un cliente donde todos los desarrolladores estaban usando el mismo IDE. Su administrador de sistemas daba a cada desarrollador un conjunto de instrucciones para instalar paquetes adicionales al IDE. Estas instrucciones ocupaban muchas páginas (páginas llenas de “haga clic aquí”, “desplace allí”, “arrastre esto”, “haga doble clic en aquello” y “hágalo de nuevo”). No era de extrañar que la máquina de cada desarrollador estuviese cargada de manera un poco distinta. Se producían diferencias sutiles en el comportamiento de la aplicación cuando desarrolladores distintos ejecutaban el mismo código. Aparecían fallos en un ordenador, pero no en otros. Rastrear las diferencias en las versiones de un componente solía revelar una sorpresa.

Truco 95. No use procedimientos manuales.

Las personas no son tan repetibles como los ordenadores, ni deberíamos esperar que lo sean. Un *script* del intérprete de comandos o un programa ejecutarán las mismas instrucciones, en el mismo orden, una y otra vez.

Está en el propio control de versiones, así que también podemos examinar cambios en los procedimientos de construcción/liberación con el tiempo (“pero antes funcionaba...”).

Todo depende de la automatización. No puede construir el proyecto en un servidor de la nube anónimo a menos que la construcción sea completamente automática. No puede desplegar de manera automática si hay pasos manuales involucrados. Y, una vez que introduzca pasos manuales (“solo para esta parte...”), habrá roto una ventana muy grande.[9]

Con estos tres pilares del control de versiones, las pruebas despiadadas y la automatización completa, su proyecto tendrá los cimientos sólidos que necesita para poder concentrarse en la parte difícil: deleitar a los usuarios.

Las secciones relacionadas incluyen

- Tema 11, “Reversibilidad”.
- Tema 12, “Balas trazadoras”.
- Tema 17, “Jugar con el intérprete de comandos”.
- Tema 19, “Control de versiones”.
- Tema 41, “Probar para escribir código”.
- Tema 49, “Equipos pragmáticos”.
- Tema 50, “Los cocos no sirven”.

Retos

- ¿Son automáticas sus construcciones nocturnas o continuas, pero el despliegue a producción no? ¿Por qué? ¿Qué tiene de especial ese servidor?
- ¿Puede probar de manera automática su proyecto por completo? Muchos equipos se ven obligados a responder “no”. ¿Por qué? ¿Es demasiado difícil definir los resultados aceptables? ¿No hará esto que sea difícil demostrar a los patrocinadores que el proyecto está “acabado”?
- ¿Es demasiado difícil probar la lógica de la aplicación independientemente de la GUI? ¿Qué dice esto acerca de la GUI? ¿Qué hay del acoplamiento?

52 Deleite a sus usuarios

Cuando cautivamos a las personas, el objetivo no es sacar dinero de ellas o conseguir que hagan lo que queremos, sino producirles un gran deleite.

—Guy Kawasaki

Nuestro objetivo como desarrolladores es deleitar a los usuarios. Para eso estamos aquí. No para extraer sus datos, conseguir que visiten nuestro sitio o vaciar sus carteras. Objetivos perversos aparte, ni siquiera entregar a tiempo software que funcione es suficiente. Eso, por sí solo, no les satisfará.

A nuestros usuarios no les motiva especialmente el código, sino que tienen un problema de negocio que necesita resolverse dentro del contexto de sus objetivos y su presupuesto. Tienen la creencia de que, trabajando con nuestro equipo, conseguirán hacerlo.

Sus expectativas no están relacionadas con el software. Ni siquiera están implícitas en las especificaciones que nos dan (porque esas especificaciones estarán incompletas hasta que nuestro equipo haya pasado por ellas junto a ellos varias veces).

¿Cómo descubrimos sus expectativas, entonces? Hagámonos una simple pregunta:

¿Cómo sabremos que todos hemos tenido éxito un mes (o un año, o lo que sea) después de que este proyecto haya acabado?

Puede que le sorprenda la respuesta. Un proyecto para mejorar las recomendaciones de un producto podría juzgarse según la retención de clientes; un proyecto para consolidar dos bases de datos podría juzgarse según la calidad de los datos, o podría tener que ver con el ahorro en los costes. Pero son estas expectativas de negocio las que cuentan de verdad, no solo el proyecto de software en sí. El software es solo un medio para lograr estos fines.

Y ahora que hemos desenterrado algunas de las expectativas subyacentes detrás del proyecto, podemos empezar a pensar en cómo cumplirlas:

- Asegúrese de que todos los miembros del equipo tienen claras estas expectativas.

- Cuando tome decisiones, piense en qué ruta hacia delante se acerca más a esas expectativas.
- Analice de manera crítica los requisitos del usuario a la luz de las expectativas. En muchos proyectos hemos descubierto que el "requisito" afirmado era en realidad solo una suposición de lo que podía hacer la tecnología: era un plan de implementación *amateur* disfrazado de documento de requisitos. No tenga miedo de hacer sugerencias que cambien el requisito si puede demostrar que el proyecto se acerca al objetivo.
- Siga pensando en estas expectativas a medida que vaya avanzando en el proyecto.

Hemos descubierto que, a medida que crece nuestro conocimiento del dominio, se nos da mejor hacer sugerencias acerca de otras cosas que podrían hacerse para abordar los problemas subyacentes del negocio. Creemos firmemente que los desarrolladores, que están expuestos a muchos aspectos diferentes de una organización, a menudo ven formas de entrelazar partes distintas del negocio que no siempre son obvias para los departamentos individuales.

Truco 96. Deleite a los usuarios, no entregue código sin más.

Si quiere deleitar a su cliente, forje con él una relación donde pueda ayudar de manera activa a resolver sus problemas. Incluso aunque su puesto puede ser alguna variación de "Desarrollador de software" o "Ingeniero de software", la verdad es que deberá ser "Solucionador de problemas". Eso es lo que hacemos, y esa es la esencia de un programador pragmático.

Resolvemos problemas.

Las secciones relacionadas incluyen

- Tema 12, "Balas trazadoras".
- Tema 13, "Prototipos y notas en *post-its*".
- Tema 45, "El pozo de los requisitos".

53 Orgullo y prejuicio

Nos has deleitado ya bastante.

—Jane Austen, Orgullo y prejuicio

Los programadores pragmáticos no eludimos la responsabilidad, sino que nos alegramos de aceptar retos y de lograr que se conozca nuestra pericia. Si somos responsables de un diseño o una porción de código, hacemos un trabajo del que podemos estar orgullosos.

Truco 97. Firme su trabajo.

Los antiguos artesanos se enorgullecían de firmar su trabajo. Usted también debería hacerlo.

Sin embargo, los equipos de proyecto están formados por personas, y esta regla puede causar problemas. En algunos proyectos, la idea de la propiedad del código puede generar problemas de cooperación. La gente puede volverse territorial o estar poco dispuesta a trabajar en elementos básicos comunes. El proyecto puede acabar convirtiéndose en un puñado de pequeños feudos aislados. Acabará prejuzgando en favor de su código y contra el de sus compañeros.

Eso no es lo que queremos. No debería defender con celo su código frente a intrusos; por la misma razón, debería tratar el código de otras personas con respeto. La regla de oro (“Trate a los demás como quiera que le traten a usted”) y unos cimientos de respeto mutuo entre los desarrolladores son cruciales para que este truco funcione.

El anonimato, sobre todo en proyectos grandes, puede ser caldo de cultivo para chapuzas, errores, pereza y mal código. Se vuelve demasiado fácil vernos a nosotros mismos solo como una pieza más del engranaje, ofreciendo malas excusas en informes de estado infinitos en vez de buen código.

Aunque el código tiene que ser propio, no tiene por qué ser propiedad de un individuo. De hecho, la Programación Extrema[10] de Kent Beck recomienda la propiedad comunitaria del código (pero esto también

requiere prácticas adicionales, como la programación en pareja, para proteger frente a los peligros del anonimato).

Queremos ver el orgullo de la propiedad. “Yo escribí esto, y doy la cara por mi trabajo”. Su firma debería llegar a reconocerse como un indicador de calidad. La gente debería ver su nombre en un código y esperar que sea sólido, que esté bien escrito, probado y documentado. Un trabajo profesional de verdad. Escrito por un profesional.

Un programador pragmático.

Gracias.

---

[1] A medida que crece el tamaño de un equipo, las rutas de comunicación crecen a una proporción de $O(n^2)$, donde n es el número de miembros del equipo. En equipos más grandes, la comunicación empieza a fallar y se vuelve inefectiva.

[2] Un gráfico *burn up* es mejor para esto que los gráficos *burn down*, que son más habituales. Con un gráfico *burn up*, podemos ver con claridad cómo las características adicionales cambian las reglas del juego.

[3] El equipo habla con una sola voz a nivel externo. A nivel interno, recomendamos encarecidamente el debate animado y sólido. Los buenos desarrolladores tienden a ser apasionados sobre su trabajo.

[4] Andy ha conocido a equipos que realizan sus reuniones Scrum diarias los viernes.

[5] Consulte https://es.wikipedia.org/wiki/Culto_cargo.

[6] Hemos visto esto con nuestros propios ojos más veces de las que se imaginaría.

[7] https://netflix.github.io/chaosmonkey.

[8] Para ver un estudio interesante de la correlación entre la cobertura de pruebas y los defectos, consulte *Mythical Unit Test Coverage* [ADSS18].

[9] Recuerde siempre la "Entropía del software". Siempre.

[10] http://www.extremeprogramming.org.

Posfacio

A la larga, damos forma a nuestras vidas y nos damos forma a nosotros mismos. El proceso no termina hasta que morimos. Y las decisiones que tomamos son, en última instancia, nuestra propia responsabilidad.

—Eleanor Roosevelt

En los veinte años anteriores a la primera edición, fuimos parte de la evolución de los ordenadores de una curiosidad periférica a un imperativo moderno para los negocios. En los veinte años posteriores, el software ha crecido más allá de las simples máquinas empresariales y ha pasado a dominar el mundo. Pero ¿qué significa eso para nosotros?

En *The Mythical Man-Month: Essays on Software Engineering* [Bro96], Fred Brooks decía: “El programador, como el poeta, trabaja solo ligeramente alejado del pensamiento puro. Construye sus castillos en el aire, a partir del aire, creando con el esfuerzo de la imaginación”. Empezamos con una página en blanco y podemos crear casi cualquier cosa que podamos imaginar. Y las cosas que creamos pueden cambiar el mundo.

Desde Twitter ayudando a la gente a planear revoluciones, al procesador del coche funcionando para evitar un derrape, al *smartphone* que hace que ya no tengamos que recordar detalles cotidianos molestos, nuestros programas están por todas partes. Nuestra imaginación está por todas partes.

Los desarrolladores somos unos privilegiados. Estamos construyendo de verdad el futuro. Es una cantidad extraordinaria de poder. Y ese poder conlleva una extraordinaria responsabilidad.

¿Con qué frecuencia nos paramos a pensar en ello? ¿Con qué frecuencia hablamos, entre nosotros y con un público más general, de lo que esto significa?

Los dispositivos integrados utilizan un orden de magnitud más de ordenadores que aquellos utilizados en portátiles, ordenadores de sobremesa y centros de datos. A menudo, estos ordenadores integrados controlan

sistemas vitales, desde centrales eléctricas a coches y equipos médicos. Incluso un simple sistema de control de calefacción central o un electrodoméstico puede matar a alguien si está mal diseñado o implementado. Cuando desarrollamos para estos dispositivos, aceptamos una responsabilidad abrumadora.

Muchos sistemas no integrados también pueden hacer un gran bien o un gran daño. Las redes sociales pueden promover una revolución pacífica o fomentar un odio atroz. Los macrodatos pueden hacer que las compras sean más fáciles y pueden destruir cualquier vestigio de privacidad que crea que tiene. Los sistemas de banca toman decisiones sobre préstamos que pueden cambiar la vida de las personas, y casi cualquier sistema puede utilizarse para cotillear en los asuntos de sus usuarios.

Hemos visto indicios de las posibilidades de un futuro utópico y ejemplos de consecuencias no planeadas que llevan a distopías de pesadilla. La diferencia entre los dos resultados podría ser más sutil de lo que piensa. Y todo está en sus manos.

La brújula moral

El precio de este poder inesperado es la vigilancia. Nuestras acciones afectan a las personas de manera directa. Ya no se trata del programa por hobby en la CPU de 8 bits en el garaje, el proceso de negocio por lotes aislado en el *mainframe* del centro de datos, ni siquiera del ordenador de sobremesa; nuestro software entrelaza el tejido mismo de la vida moderna diaria.

Tenemos el deber de hacernos a nosotros mismos dos preguntas sobre cada código que entreguemos:

1. ¿He protegido al usuario?
2. ¿Usaría yo esto?

Primero, debería preguntarse: “¿He hecho todo lo posible por proteger a los usuarios de este código de daños?”. ¿He tomado las medidas preventivas necesarias para aplicar parches de seguridad en curso en ese sencillo monitor para bebés? ¿Me he asegurado de que falle como falle el

termostato de la calefacción central automático el cliente seguirá teniendo control manual? ¿Estoy almacenando solo los datos que necesito y encriptando cualquier cosa personal?

Nadie es perfecto; a todos se nos pasan cosas por alto de vez en cuando. Pero, si no puede afirmar con sinceridad que ha intentado hacer una lista de todas las consecuencias y se ha asegurado de proteger de ellas a los usuarios, entonces tiene algo de responsabilidad cuando las cosas salen mal.

Truco 98. Primero, no cause daño.

En segundo lugar, hay un criterio relacionado con la regla de oro: ¿estaría yo satisfecho si fuese usuario de este software? ¿Quiero que se comparta mi información personal? ¿Quiero que mis movimientos se pasen a puntos de venta? ¿Estaría tranquilo si me llevase este vehículo autónomo? ¿Estoy cómodo haciendo esto?

Algunas ideas ingeniosas empiezan a bordear los límites del comportamiento ético y, si usted está involucrado en ese proyecto, es tan responsable como los patrocinadores.

No importa cuántos grados de separación quiera racionalizar, hay una regla que sigue siendo cierta:

Truco 99. No dé alas a los capullos.

Imagine el futuro que quiere

Depende de usted. Son su imaginación, sus esperanzas y sus preocupaciones las que proporcionan el pensamiento puro que construye los próximos veinte años y más.

Está construyendo el futuro, para usted y para sus descendientes. Su deber es hacer que sea un futuro en el que todos querríamos vivir. Reconozca cuándo está haciendo algo que va contra ese ideal y tenga el coraje de decir “¡no!”. Visualice el futuro que podríamos tener y tenga el valor de crearlo. Construya castillos en el aire todos los días.

Tenemos una vida asombrosa.

Truco 100. Es su vida.
Compártala. Celébrela. Construýala.
¡Y DIVIÉRTASE!

Bibliografía

[ADSS18] Vard Antinyan, Jesper Derehag, Anna Sandberg, y Miroslaw Staron. “Mythical Unit Test Coverage”. *IEEE Software*. 35:73-79, 2018.

[And10] Jackie Andrade. “What does doodling do?”. *Applied Cognitive Psychology*. 24(1):100-106, 2010, enero.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, Carolina del Norte, 2007.

[BR89] Albert J. Bernstein y Sydney Craft Rozen. *Cerebros de dinosaurio. Cómo tratar con personas imposibles en el trabajo*. Editorial Diana, 1992.

[Bro96] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, Aniversario, 1996.

[CN91] Brad J. Cox y Andrew J. Novobilski. *Programación orientada a objetos: un enfoque evolutivo*. Addison-Wesley/Díaz de Santos, Segunda, 1993.

[Con68] Melvin E. Conway. “How do Committees Invent?” *Datamation*. 14(5):28-31, 1968, abril.

[de 98] Gavin de Becker. *El valor del miedo: señales de alarma que nos protegen de la violencia*. Urano, 1999.

[DL13] Tom DeMacro y Tim Lister. *Peopleware: Productive Projects and Teams*. Addison-Wesley, Boston, MA, Tercera, 2013.

[Fow00] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, MA, Segunda, 2000.

[Fow04] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, MA, Tercera, 2004.

[Fow19] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, Segunda, 2019.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Patrones de Diseño. Elementos de software orientado a objetos reutilizable*. Pearson Addison-Wesley, Madrid, 2002.

[Hol92] Michael Holt. *Math Puzzles & Games*. Dorset House, Nueva York, NY, 1992.

[Hun08] Andy Hunt. *Pragmatic Thinking and Learning: Refactor Your Wetware*. The Pragmatic Bookshelf, Raleigh, Carolina del Norte, 2008.

[Joi94] T.E. Joiner. “Contagious depression: Existence, specificity to depressed

symptoms, and the role of reassurance seeking". *Journal of Personality and Social Psychology*. 67(2):287-296, 1994, agosto.

[Knu11] Donald E. Knuth. *The Art of Computer Programming, Volumen 4A: Combinatorial Algorithms, Parte 1*. Addison-Wesley, Boston, MA, 2011.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volumen 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, Tercera, 1998.

[Knu98a] Donald E. Knuth. *The Art of Computer Programming, Volumen 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, Tercera, 1998.

[Knu98b] Donald E. Knuth. *The Art of Computer Programming, Volumen 3: Sorting and Searching*. Addison-Wesley, Reading, MA, Segunda, 1998.

[KP99] Brian W. Kernighan y Rob Pike. *La práctica de la programación*. Pearson Educación, México, D.F., 2000.

[Mey97] Bertrand Meyer. *Construcción de software orientado a objetos*. Prentice Hall, Madrid, 1998.

[Mul18] Jerry Z. Muller. *The Tyranny of Metrics*. Princeton University Press, Princeton NJ, 2018.

[SF13] Robert Sedgewick y Phillipe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Boston, MA, Segunda, 2013.

[Str35] James Ridley Stroop. "Studies of Interference in Serial Verbal Reactions". *Journal of Experimental Psychology*. 18:643-662, 1935.

[SW11] Robert Sedgewick y Kevin Wayne. *Algorithms*. Addison-Wesley, Boston, MA, Cuarta, 2011.

[Tal10] Nassim Nicholas Taleb. *El cisne negro: El impacto de lo altamente improbable*. Paidos Ibérica, 2011.

[WH82] James Q. Wilson y George Helling. "The police and neighborhood safety". *The Atlantic Monthly*. 249[3]:29-38, 1982, marzo.

[YC79] Edward Yourdon y Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, NJ, 1979.

[You95] Edward Yourdon. "When good-enough software is best". *IEEE Software*. 1995, mayo.

Posibles respuestas a los ejercicios

Respuesta ejercicio 1

A nuestro entender, la clase `Split2` es más ortogonal. Se concentra en su propia tarea, dividir líneas, e ignora detalles como de dónde vienen las líneas. Eso no solo hace que el código sea más fácil de desarrollar, sino que también lo hace más flexible. `Split2` puede dividir líneas leídas desde un archivo, generadas por otra rutina o pasadas a través del entorno.

Respuesta ejercicio 2

Vamos a empezar con una afirmación: puede escribir código bueno y ortogonal en casi cualquier lenguaje. Al mismo tiempo, cada lenguaje tiene tentaciones: características que pueden llevar a un incremento del acoplamiento y una reducción de la ortogonalidad.

En los lenguajes orientados a objetos, características como la herencia múltiple, las excepciones, la sobrecarga del operador y la anulación del método padre (mediante subclase) ofrecen muchas oportunidades para aumentar el acoplamiento de formas que no son evidentes. También hay un tipo de acoplamiento porque una clase acopla el código a los datos. Por lo general, eso es algo bueno (cuando el acoplamiento es bueno, lo llamamos cohesión). Pero, si no hace que sus clases se concentren lo suficiente, puede llevar a unas interfaces bastante feas.

En los lenguajes funcionales, se le anima a escribir muchas funciones pequeñas desacopladas y combinarlas de maneras diferentes para resolver un problema. En teoría, esto suena bien. En la práctica, a menudo es bueno. Pero aquí también puede darse una forma de acoplamiento. Estas funciones suelen transformar datos, lo que significa que el resultado de una función puede convertirse en la entrada de otra. Si no tiene cuidado, hacer un cambio en el formato de los datos que genera una función puede causar un

fallo en alguna otra parte más adelante en el flujo de transformaciones. Los lenguajes con buenos sistemas de tipos pueden ayudar a paliar esto.

Respuesta ejercicio 3

¡La baja tecnología al rescate! Haga algunos dibujos con el rotulador en la pizarra: un coche, un teléfono y una casa. No hace falta que sean obras de arte; el contorno con palitos sirve. Ponga notas en *post-its* que describan los contenidos de las páginas de destino en las áreas en las que se pueda hacer clic. A medida que progrese la reunión, puede perfeccionar los dibujos y las ubicaciones de los *post-its*.

Respuesta ejercicio 4

Puesto que queremos que el lenguaje sea extensible, haremos que el analizador esté gestionado por una tabla. Cada entrada de la tabla contiene la letra del comando, una bandera para indicar si se requiere un argumento y el nombre de la rutina a la que hay que llamar para manejar ese comando en particular.

lang/turtle.c

```
typedef struct {
   char cmd; /* la letra del comando */
   int hasArg; /* ¿toma un argumento? */
   void (*func)(int, int); /* rutina a la que hay que llamar */
} Command;

static Command cmds[] = {
   { 'P', ARG, doSelectPen },
   { 'U', NO_ARG, doPenUp },
   { 'D', NO_ARG, doPenDown },
   { 'N', ARG, doPenDir },
   { 'E', ARG, doPenDir },
   { 'S', ARG, doPenDir },
   { 'W', ARG, doPenDir }
};
```

El programa principal es bastante simple: leer una línea, buscar el comando, tomar el argumento si es necesario y, después, llamar a la función

manejadora.

lang/turtle.c

```
while (fgets(buff, sizeof(buff), stdin)) {

   Command *cmd = findCommand(*buff);

   if (cmd) {
      int arg = 0;

      if (cmd->hasArg && !getArg(buff+1, &arg)) {
         fprintf(stderr, "'%c' needs an argument\n", *buff);
         continue;
      }

      cmd->func(*buff, arg);
   }
}
```

La función que busca un comando realiza una búsqueda lineal de la tabla, devolviendo la entrada correspondiente o NULL.

lang/turtle.c

```
Command *findCommand(int cmd) {
   int i;

   for (i = 0; i < ARRAY_SIZE(cmds); i++) {
      if (cmds[i].cmd == cmd)
         return cmds + i;
   }

   fprintf(stderr, "Unknown command '%c'\n", cmd);
   return 0;
}
```

Por último, leer el argumento numérico es bastante sencillo usando `sscanf`.

lang/turtle.c

```
int getArg(const char *buff, int *result) {
   return sscanf(buff, "%d", result) == 1;
}
```

Respuesta ejercicio 5

En realidad, ya ha resuelto este problema en el ejercicio anterior, donde ha escrito un intérprete para el lenguaje externo, contendrá el intérprete interno. En el caso de nuestro código de muestra, esto son las funciones doXxx.

Respuesta ejercicio 6

Usando BNF, una especificación de tiempo podría ser

tiempo ::= *hora ampm* | *hora* : *minuto ampm* | *hora* : *minuto*
ampm ::= am | pm
hora ::= *dígito* | *dígito dígito*
minuto ::= *dígito dígito*
dígito ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Una definición mejor de hora y minuto tendría en cuenta que una hora solo puede estar entre 00 y 23, y un minuto, entre 00 y 59:

hora ::= *decenas h dígito* | *dígito*
minuto ::= *decenas m dígito*
decenas h ::= 0 | 1
decenas m ::= 0 | 1 | 2 | 3 | 4 | 5
dígito ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Respuesta ejercicio 7

Este es el analizador sintáctico escrito con la biblioteca Pegjs JavaScript:

lang/peg_parser/time_parser.pegjs

```
time
  = h:hour offset:ampm { return h + offset }
  / h:hour ":" m:minute offset:ampm { return h + m + offset }
  / h:hour ":" m:minute { return h + m }

ampm
```

```
   = “am” { return 0 }
   / “pm” { return 12*60 }

hour
   = h:two_hour_digits { return h*60 }
   / h:digit { return h*60 }

minute
   = d1:[0-5] d2:[0-9] { return parseInt(d1+d2, 10); }

digit
   = digit:[0-9] { return parseInt(digit, 10); }

two_hour_digits
   = d1:[01] d2:[0-9 ] { return parseInt(d1+d2, 10); }
   / d1:[2] d2:[0-3] { return parseInt(d1+d2, 10); }
```

Las pruebas lo muestran en uso:

lang/peg_parser/test_time_parser.js

```
let test = require(‘tape’);
let time_parser = require(‘./time_parser.js’);

// tiempo ::= hora ampm |
// hora : minuto ampm |
// hora : minuto
//
// ampm ::= am | pm
//
// hora ::= dígito | dígito dígito
//
// minuto ::= dígito dígito
//
// dígito ::= 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

const h = (val) => val*60;
const m = (val) => val;
const am = (val) => val;
const pm = (val) => val + h(12);

let tests = {

   “1am”: h(1),
   “1pm”: pm(h(1)),

   “2:30”: h(2) + m(30),
   “14:30”: pm(h(2)) + m(30),
```

```
   “2:30pm”: pm(h(2)) + m(30),
}

test(‘time parsing’, function (t) {
   for (const string in tests) {
      let result = time_parser.parse(string)
      t.equal(result, tests[string], string);
   }
   t.end()
});
```

Respuesta ejercicio 8

Aquí hay una solución posible en Ruby:

lang/re_parser/time_parser.rb

```
TIME_RE = %r{
(?<digit>[0-9]){0}
(?<h_ten>[0-1]){0}
(?<m_ten>[0-6]){0}
(?<ampm> am | pm){0}
(?<hour> (\g<h_ten> \g<digit>) | \g<digit>){0}
(?<minute> \g<m_ten> \g<digit>){0}

\A(
     ( \g<hour> \g<ampm> )
   | ( \g<hour> : \g<minute> \g<ampm> )
   | ( \g<hour> : \g<minute> )
)\Z

}x

def parse_time(string)
   result = TIME_RE.match(string)
   if result
      result[:hour].to_i * 60 +
      (result[:minute] || “0”).to_i +
      (result[:ampm] == “pm” ? 12*60 : 0)
   end
end
```

(Este código utiliza el truco de definir patrones con nombre al principio de las expresiones regulares y, después, referenciarlos como subpatrones en la correspondencia real).

Respuesta ejercicio 9

Nuestra respuesta debe formularse en varias asunciones:

- El dispositivo de almacenamiento contiene la información que necesitamos que se transfiera.
- Conocemos la velocidad a la que camina la persona.
- Conocemos la distancia entre los ordenadores.
- Nos estamos dando cuenta del tiempo que se tarda en transferir información a y desde el dispositivo de almacenamiento.
- La tara de almacenar los datos es más o menos igual a la tara de enviarlos a través de una línea de comunicación.

Respuesta ejercicio 10

Sujeto a las advertencias de la respuesta anterior: una cinta de 1 TB contiene 8×240, o 243 bits, así que una línea de 1 Gbps tendría que bombear datos durante aproximadamente 9.000 segundos, o alrededor de 2½ horas, para transferir la cantidad equivalente de información. Si la persona camina a una velocidad constante de 5,6 km/h, entonces los dos ordenadores tendrían que estar a casi 14,5 km de distancia el uno del otro para que la línea de comunicaciones superase el rendimiento de nuestra mensajera. De lo contrario, gana la persona.

Respuesta ejercicios 11-13

Se trata de ejercicios de respuesta abierta para los que no especificamos qué lenguaje o conjunto de herramientas utilizar, ya que depende del lector. Hay un número infinito de respuestas correctas.

Respuesta ejercicio 14

Vamos a mostrar las firmas de función en Java, con las precondiciones y postcondiciones en comentarios.

Primero, la invariante para la clase:

```
/**
   * @invariant getSpeed() > 0
   * implies isFull() // No poner en marcha vacía
   *
   * @invariant getSpeed() >= 0 &&
   * getSpeed() < 10 // Comprobación de rango
   */
```

A continuación, las precondiciones y postcondiciones:

```
/**
   * @pre Math.abs(getSpeed()-x) <= 1 // Solo cambia de uno en uno
   * @pre x >= 0 && x < 10 // Comprobación de rango
   * @post getSpeed() == x // Respetar la velocidad solicitada
   */
public void setSpeed(final int x)

/**
   * @pre !isFull() // No llenarla dos veces
   * @post isFull() // Asegurarse de que ha acabado
   */
void fill()

/**
   * @pre isFull() // No vaciarla dos veces
   * @post !isFull() // Asegurarse de que ha acabado
   */
void empty()
```

Respuesta ejercicio 15

Hay 21 términos en la serie. Si ha dicho 21, ha experimentado un "error por uno" (en inglés, este tipo de error se conoce como "*fence-post error*", error de poste de cerca, como si al construir una cerca no supiese si contar los postes o los espacios entre ellos).

Respuesta ejercicio 16

- El mes de septiembre de 1752 solo tuvo 19 días. Esto se hizo para sincronizar los calendarios como parte de la reforma gregoriana.
- El directorio podría haber sido eliminado por otro proceso, podría ser que usted no tuviese permiso para leerlo, la unidad podría no estar montada...; ya se hace una idea.

- No hemos especificado los tipos de `a` y `b` de manera intencionada. La sobrecarga de operadores podría haber definido `+`, `=`, o `!=` para tener un comportamiento inesperado. Además, `a` y `b` podrían ser alias para la misma variable, de forma que la segunda asignación sobrescribirá el valor almacenado en la primera. Además, si el programa es concurrente y está mal escrito, `a` podría haberse actualizado para cuando se realice la suma.
- En geometría no euclidiana, la suma de los ángulos de un triángulo no será 180°. Piense en un triángulo trazado sobre la superficie de una esfera.
- Los minutos intercalares pueden tener 61 o 62 segundos.
- Dependiendo del lenguaje, el desbordamiento numérico puede dejar el resultado de `a+1` negativo.

Respuesta ejercicio 17

En la mayoría de las implementaciones de C y C++, no hay forma de comprobar si un puntero apunta de verdad a memoria válida. Un error común es desasignar un bloque de memoria y hacer referencia a esa memoria más tarde en el programa. Para entonces, puede que la memoria a la que se apuntaba ya se haya reasignado a algún otro propósito. Al establecer el puntero en `NULL`, el programador espera evitar esas referencias falsas; en la mayoría de los casos, quitar la referencia de un puntero `NULL` generará un error en tiempo de ejecución.

Respuesta ejercicio 18

Al establecer la referencia en `NULL`, se reduce el número de punteros al objeto referenciado en uno. Una vez que la cuenta llegue a cero, el objeto será elegible para el recolector de basura. Establecer las referencias en `NULL` puede ser significativo para los programas de larga duración, donde los programadores necesitan asegurarse de que la utilización de la memoria no se incrementa con el tiempo.

Respuesta ejercicio 19

Una implementación simple podría ser:

event/strings_ex_1.rb

```
class FSM
   def initialize(transitions, initial_state)
     @transitions = transitions
     @state = initial_state
   end
   def accept(event)
     @state,     action     =     TRANSITIONS[@state][event]     ||
TRANSITIONS[@state][:default]
   end
end
```

Respuesta ejercicio 20

- ...tres eventos de caída de interfaz de red en cinco minutos. Esto podría implementarse utilizando una máquina de estados, pero sería más complicado de lo que podría parecer en principio: si recibe eventos en los minutos 1, 4, 7 y 8, entonces debería activar la alerta en el cuarto evento, lo que significa que la máquina de estados necesita ser capaz de manejar el reinicio por sí misma.
  Por esta razón, los *streams* de eventos parecerían ser la tecnología elegida. Hay una función llamada `buffer` con parámetros `size` y `offset` que le dejaría devolver cada grupo de tres eventos entrantes. Entonces, podría fijarse en los sellos de tiempo del primer y el último evento de un grupo para determinar si la alarma debería activarse.
- ... después del atardecer, y se detecta movimiento en la parte inferior de las escaleras, seguido de movimiento detectado en la parte superior de las escaleras... Es probable que esto pudiera implementarse utilizando una combinación de pubsub y máquinas de estados. Podría utilizar pubsub para diseminar eventos a cualquier cantidad de máquinas de estados y, después, hacer que las máquinas de estados determinen lo que hay que hacer.
- ... notificar a varios sistemas de creación de informes que se ha completado un pedido. Es probable que la mejor forma de manejar esto sea utilizar pubsub. Podría querer utilizar *streams*, pero eso

requeriría que los sistemas a los que se va a notificar estuviesen también basados en *streams*.

- ... tres servicios *backend* y esperar las respuestas. Esto es similar a nuestro ejemplo que utilizaba *streams* para extraer datos de usuarios.

Respuesta ejercicio 21

1. Los gastos de envío y el impuesto sobre las ventas se añaden a un pedido:

```
pedido básico ▶ pedido finalizado
```

   En el código convencional, es probable que tuviese una función que calculase los gastos de envío y otra que calculase el impuesto, pero aquí estamos pensando en transformaciones, así que transformamos un pedido solo con ítems en un nuevo tipo de cosa: un pedido que puede enviarse.

2. Su aplicación carga información de configuración desde un archivo con nombre:

```
nombre de archivo ▶ estructura de configuración
```

3. Alguien inicia sesión en una aplicación web:

```
credenciales de usuario ▶ sesión
```

Respuesta ejercicio 22

La transformación de alto nivel:

```
contenidos del campo como cadena
▶ [validar & convertir]
▶ {:ok, valor} | {:error, motivo}
```

podría descomponerse en:

```
contenidos del campo como cadena
▶ [convertir cadena a entero]
▶ [comprobar valor >= 18]
▶ [comprobar valor <= 150]
▶ {:ok, valor} | {:error, motivo}
```

Esto da por hecho que tiene una *pipeline* para el manejo de errores.

Respuesta ejercicio 23

Vamos a responder primero a la segunda parte: nosotros preferimos el primer fragmento de código.

En la segunda porción de código, cada paso devuelve un objeto que implementa la siguiente función a la que llamamos: el objeto devuelto por `content_of` debe implementar `find_matching_lines`, y así sucesivamente.

Esto significa que el objeto devuelto por `content_of` está acoplado a nuestro código. Imagine que cambia el requisito y tenemos que ignorar las líneas que empiezan con un carácter #. En el estilo de las transformaciones, eso sería fácil:

```
const content = File.read(file_name);
const no_comments = remove_comments(content)
const lines = find_matching_lines(no_comments, pattern)
const result = truncate_lines(lines)
```

Podríamos incluso invertir el orden de `remove_comments` y `find_matching_lines` y aun así funcionaría. Pero, en el estilo encadenado, eso sería más difícil. ¿Dónde debería vivir nuestro método `remove_comments`: en el objeto devuelto por `content_of` o en el objeto devuelto por `find_matching_lines`? ¿Y qué otro código se estropeará si cambiamos ese objeto? Este acoplamiento es la razón por la que el estilo de encadenamiento de métodos se llama a veces “choque de trenes” (*train wreck*).

Respuesta ejercicio 24

- **Procesamiento de imágenes:** Para la programación sencilla de una carga de trabajo entre los procesos paralelos, una cola de trabajo compartida puede ser más que adecuada. Puede que quiera considerar el uso de un sistema de pizarra si hay retroalimentación, es decir, si los resultados de una porción procesada afectan a otras porciones,

como en aplicaciones de visión artificial o transformaciones complejas de distorsión de imágenes 3D.

- **Agenda para grupos:** Aquí podría ser una solución adecuada. Puede publicar reuniones programadas y disponibilidad en la pizarra. Tiene entidades que funcionan de manera autónoma, el *feedback* de las decisiones es importante y los participantes pueden ir y venir.
  Quizá le convendría dividir este tipo de sistema de pizarra dependiendo de quién esté buscando: puede que al personal junior solo le interese la oficina inmediata, tal vez el departamento de recursos humanos quiera solo las oficinas de habla inglesa de todo el mundo y el CEO podría querer acceso a todo.
  También hay cierta flexibilidad en el formato de los datos: somos libres de ignorar formatos o lenguajes que no entendamos. Tenemos que entender formatos diferentes solo para aquellas oficinas que tienen reuniones entre sí y no necesitamos exponer a todos los participantes a una clausura transitiva completa de todos los formatos posibles. Esto reduce el acoplamiento a las partes donde sea necesario y no nos restringe de manera artificial.
- **Herramienta de monitorización de red:** Esto es muy similar al programa de solicitud de hipotecas/préstamos. Tenemos informes de problemas enviados por usuarios y estadísticas de las que se informa de manera automática, todo ello publicado en la pizarra. Un agente humano o de software puede analizar la pizarra para diagnosticar los fallos de red: dos errores en una línea podrían ser solo rayos cósmicos, pero 20.000 errores significan un problema de hardware. Del mismo modo que los detectives resuelven el misterio de un asesinato, podemos tener múltiples entidades analizando y aportando ideas para resolver los problemas de la red.

Respuesta ejercicio 25

La asunción con una lista de pares clave-valor es, por lo general, que la clave es única y las bibliotecas de *hash* suelen imponerlo, ya sea mediante el comportamiento del propio *hash* o con mensajes de error explícitos para claves duplicadas. Sin embargo, una matriz no suele tener estas restricciones y almacenará alegremente claves duplicadas a menos que

escribamos un código que le diga de forma específica que no lo haga. Así pues, en este caso, la primera clave encontrada que coincide con `DepositAccount` gana, y cualquier entrada coincidente restante se ignora. El orden de las entradas no está garantizado, así que a veces funciona y a veces, no.

¿Y qué pasa con la diferencia en las máquinas de desarrollo y producción? Es solo una coincidencia.

Respuesta ejercicio 26

El hecho de que un campo puramente numérico funcione en EE. UU., Canadá y el Caribe es una casualidad. Según las especificaciones de la UIT, el formato de las llamadas internacionales comienza por un `+` literal. El carácter `*` también se utiliza en algunos lugares y, de forma más común, puede haber ceros a la izquierda como parte del número. Nunca almacene un número de teléfono en un campo numérico.

Respuesta ejercicio 27

Depende de dónde esté. En EE. UU., las medidas de volumen se basan en el galón, que es el volumen de un cilindro de 6 pulgadas de alto y 7 pulgadas de diámetro, redondeado a la pulgada cúbica más cercana.

En Canadá, “una taza” en una recta podría significar cualquiera de estas opciones:

- 1/5 de un cuarto imperial, o 227 ml.
- 1/4 de un cuarto americano, o 236 ml.
- 16 cucharadas soperas métricas, o 240 ml.
- 1/4 de un litro, o 250 ml.

A menos que esté hablando de una arrocera, en cuyo caso “una taza” equivale a 180 ml. Esto deriva del *koku*, que era el volumen estimado de arroz seco necesario para alimentar a una persona durante un año: por lo visto, alrededor de 180 l. Las tazas para arroceras son de 1 *gŌ*, que es 1/1000 de un *koku*. Por tanto, más o menos la cantidad de arroz que una persona ingeriría en una sola comida.[1]

Respuesta ejercicio 28

Está claro que no podemos dar respuestas absolutas a este ejercicio. Sin embargo, podemos darle un par de pistas.

Si descubre que sus resultados no siguen una curva suave, puede que le convenga comprobar si hay alguna otra actividad utilizando algo de la potencia del procesador. Es probable que no obtenga buenas cifras si los procesos en segundo plano quitan ciclos a sus programas de forma periódica. Puede que también le interese comprobar la memoria: si la aplicación empieza a utilizar espacio de intercambio, el rendimiento caerá en picado.

El gráfico de la figura 1 muestra los resultados de ejecutar el código en uno de nuestros ordenadores.

Respuesta ejercicio 29

Hay un par de maneras de conseguirlo. Una es darle la vuelta al problema. Si la matriz tiene solo un elemento, no iteramos por el bucle. Cada iteración adicional duplica el tamaño de la matriz en la que podemos buscar. La fórmula general para el tamaño de la matriz es, por tanto, $n = 2^m$, donde `m` es el número de iteraciones. Si llevamos los logaritmos a la base 2 de cada lado, obtenemos $\lg n = \lg 2^m$, que según la definición de logaritmos se convierte en $\lg n = m$.

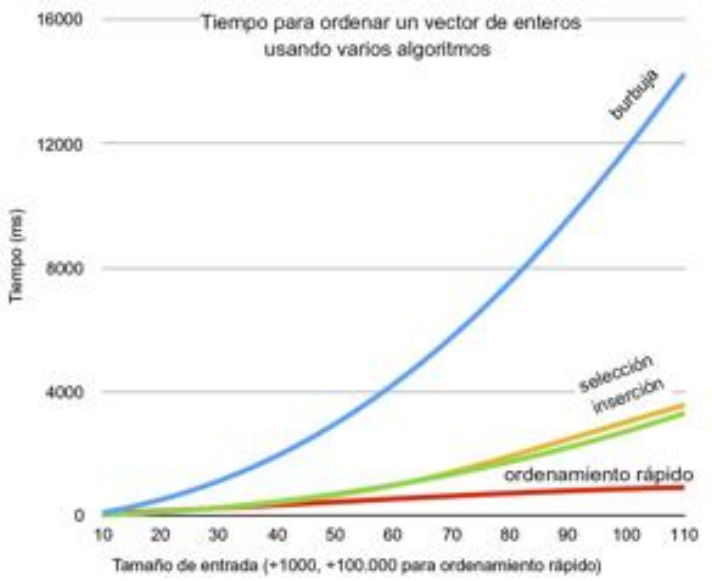


Figura 1.

Respuesta ejercicio 30

Es probable que esto recuerde demasiado a una clase de matemáticas de secundaria, pero la fórmula para convertir un logaritmo de base `a` en uno de base `b` es:

$$\log_b x = (\log_a x / \log_a b)$$

Como $\log_a b$ es una constante, podemos ignorarlo dentro de un resultado Big O.

Respuesta ejercicio 31

Una propiedad que podemos probar es que un pedido se completa con éxito si el almacén tiene suficientes artículos disponibles. Podemos generar pedidos para cantidades aleatorias de artículos y verificar que se devuelve una tupla `"OK"` si el almacén tiene *stock*.

Respuesta ejercicio 32

Este es un buen uso de las pruebas basadas en propiedades. Las pruebas unitarias pueden centrarse en casos individuales donde hemos determinado el resultado a través de otros medios y las pruebas de propiedades pueden centrarse en cosas como:

- ¿Hay dos cajas que se solapen?
- ¿Hay alguna parte de alguna caja que exceda la anchura o la longitud del camión?
- ¿Es la densidad de embalaje (área utilizada por las cajas dividida entre el área de la plataforma del camión) menor o igual a 1?
- Si es parte del requisito, ¿supera la densidad de embalaje la densidad mínima aceptable?

Respuesta ejercicio 33

1. Esta afirmación suena como un requisito real: puede que haya restricciones en la aplicación impuestas por su entorno.
2. Por sí misma, esta afirmación no es realmente un requisito, pero, para averiguar lo que se requiere en realidad, tiene que hacer la pregunta mágica: "¿Por qué?"
   Puede que sea un estándar corporativo, en cuyo caso el requisito real debería ser algo como "todos los elementos de la interfaz de usuario deben ajustarse a los Estándares para la Interfaz de Usuario MegaCorp, V12.76".
   Puede que ese sea un color que resulta que le gusta al equipo de diseño. En ese caso, debería pensar en que al equipo de diseño también le gusta cambiar de opinión y es más aconsejable expresar el requisito como "El color de fondo de todas las ventanas modales debe ser configurable. En el momento del envío, el color será gris". Sería aún mejor hacer una afirmación más amplia, como "Todos los elementos visuales de la aplicación (colores, fuentes e idiomas) deben ser configurables".
   O puede que simplemente signifique que el usuario necesita poder distinguir las ventanas modales de las no modales. En ese caso, habría que discutir la cuestión en mayor profundidad.

3. Esta afirmación no es un requisito, es arquitectura. Cuando se enfrente a algo como esto, tiene que indagar más para averiguar lo que está pensando el usuario. ¿Se trata de un problema de escala? ¿De rendimiento? ¿De costes? ¿De seguridad? Las respuestas servirán de base para su diseño.
4. Es probable que el requisito subyacente se acerque más a algo como “El sistema evitará que el usuario introduzca entradas no válidas en los campos y avisará al usuario cuando introduzca esas entradas”.
5. Esta afirmación es probablemente un requisito duro, basado en alguna limitación de hardware.

Y aquí hay una solución para el problema de los cuatro puntos:

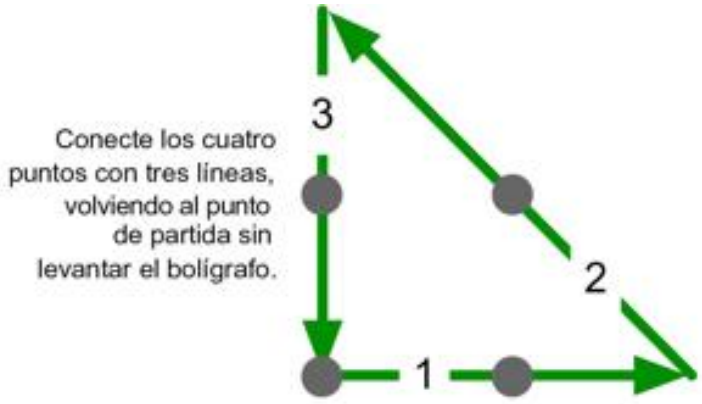


Figura 2.

---

[1] Nuestro agradecimiento por estas curiosidades a Avi Bryant (@avibryant).

Título de la obra original:
The Pragmatic Programmer: Your Journey to Mastery, 20th Anniversary Edition. 2nd edition.

Traductor:
Beatriz Pineda González

Responsable editorial:
Víctor Manuel Ruiz Calderón

Adaptación de cubierta:
Celia Antón Santos

Imagen e ilustración de cubierta:
© 2022 Mihalec/iStockphoto L.P./Guetty Images
© 2022 Idimair/iStockphoto L.P./Guetty Images

Edición en formato digital: 2022

Authorized translation from the English language edition, entitled *The Pragmatic Programmer: Your Journey to Mastery, 20th Anniversary Edition, 2nd edition*, by David Thomas and Andrew Hunt, published by Pearson Education, Inc, publishing asAddison-Wesley Professional.
Copyright © 2020 Pearson Education, Inc. All rights reserved.

© EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S. A.), 2022
Calle Juan Ignacio Luca de Tena, 15
28027 Madrid

ISBN ebook: 978-84-415-4605-9

Todos los nombres propios de programas, sistemas operativos, equipos hardware, etc. que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

Está prohibida la reproducción total o parcial de este libro electrónico, su transmisión, su descarga, su descompilación, su tratamiento informático, su almacenamiento o introducción en cualquier sistema de repositorio y recuperación, en cualquier forma o por cualquier medio, ya sea electrónico, mecánico, conocido o por inventar, sin el permiso expreso escrito de los titulares del Copyright.